

JUN 08 2008

PENT  
JUN 08 2008  
PATENT & TRADEMARK OFFICE

## UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE  
United States Patent and Trademark Office  
Address: COMMISSIONER FOR PATENTS  
P.O. Box 1450  
Alexandria, Virginia 22313-1450  
[www.uspto.gov](http://www.uspto.gov)

FD

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/711,737	09/30/2004	Lee George LABORCZFALVI	2006579-0141	5736
69665	7590	05/28/2008	EXAMINER	
CHOATE, HALL & STEWART / CITRIX SYSTEMS, INC. TWO INTERNATIONAL PLACE BOSTON, MA 02110			MORRISON, JAY A	
		ART UNIT	PAPER NUMBER	
		2168		
		MAIL DATE	DELIVERY MODE	
		05/28/2008	PAPER	

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

<b>Office Action Summary</b>	<b>Application No.</b>	<b>Applicant(s)</b>	
	10/711,737	LABORCZFALVI ET AL.	
	Examiner JAY A. MORRISON	Art Unit 2168	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

#### Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

#### Status

- 1) Responsive to communication(s) filed on 11 February 2008.
- 2a) This action is FINAL.                            2b) This action is non-final.
- 3) Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

#### Disposition of Claims

- 4) Claim(s) 1-32 is/are pending in the application.
- 4a) Of the above claim(s) \_\_\_\_\_ is/are withdrawn from consideration.
- 5) Claim(s) \_\_\_\_\_ is/are allowed.
- 6) Claim(s) 1-32 is/are rejected.
- 7) Claim(s) \_\_\_\_\_ is/are objected to.
- 8) Claim(s) \_\_\_\_\_ are subject to restriction and/or election requirement.

#### Application Papers

- 9) The specification is objected to by the Examiner.
- 10) The drawing(s) filed on \_\_\_\_\_ is/are: a) accepted or b) objected to by the Examiner.  
Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).  
Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).
- 11) The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

#### Priority under 35 U.S.C. § 119

- 12) Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).
    - a) All    b) Some \* c) None of:
      1. Certified copies of the priority documents have been received.
      2. Certified copies of the priority documents have been received in Application No. \_\_\_\_\_.
      3. Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).
- \* See the attached detailed Office action for a list of the certified copies not received.

#### Attachment(s)

- |  |   |
|--|---|
| 1) <input checked="" type="checkbox"/> Notice of References Cited (PTO-892)  | 4) <input type="checkbox"/> Interview Summary (PTO-413)           |
| 2) <input type="checkbox"/> Notice of Draftsperson's Patent Drawing Review (PTO-948)   | Paper No(s)/Mail Date. _____ .                                    |
| 3) <input checked="" type="checkbox"/> Information Disclosure Statement(s) (PTO/SB/08)<br>Paper No(s)/Mail Date <u>2/11/08</u> . | 5) <input type="checkbox"/> Notice of Informal Patent Application |
|  | 6) <input type="checkbox"/> Other: _____                          |

## **DETAILED ACTION**

### ***Remarks***

1. Claims 1-32 are pending.
2. The information disclosure statement (IDS) submitted on 2/11/2008 has been considered by the examiner.

### ***Continued Examination Under 37 CFR 1.114***

3. A request for continued examination under 37 CFR 1.114, including the fee set forth in 37 CFR 1.17(e), was filed in this application after final rejection. Since this application is eligible for continued examination under 37 CFR 1.114, and the fee set forth in 37 CFR 1.17(e) has been timely paid, the finality of the previous Office action has been withdrawn pursuant to 37 CFR 1.114. Applicant's submission filed on 2/11/2008 has been entered.

### ***Claim Rejections - 35 USC § 101***

4. 35 U.S.C. 101 reads as follows:

Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title.

5. Claims 23-32 are rejected under 35 U.S.C. 101 because the claimed invention is directed to non-statutory subject matter. The claimed subject matter, "an isolation environment", does not fit into any of the statutory categories (process, machine, manufacture, or composition of matter). In order for the claimed subject matter to fit into a statutory category under 35 USC 101, the Applicant is requested to further distinguish the claimed subject matter to adhere to one of these categories.

***Claim Rejections - 35 USC § 102***

6. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

A person shall be entitled to a patent unless –

(e) the invention was described in (1) an application for patent, published under section 122(b), by another filed in the United States before the invention by the applicant for patent or (2) a patent granted on an application for patent by another filed in the United States before the invention by the applicant for patent, except that an international application filed under the treaty defined in section 351(a) shall have the effects for purposes of this subsection of an application filed in the United States only if the international application designated the United States and was published under Article 21(2) of such treaty in the English language.

7. Claims 1-32 are rejected under 35 U.S.C. 102(e) as being anticipated by Kagi et al. ('Kagi' hereinafter) (Publication Number 2006/0064697).

As per claim 1, Kagi teaches

A method for isolating access by application programs to native resources provided by an operating system, the method comprising the steps of: (see abstract and background)

(a) redirecting to an isolation environment comprising a user isolation scope and an application isolation scope a request for a native resource made by a process executing on behalf of a first user; (virtual machine which performs isolation by virtualizing resources, paragraph [0019], lines 5-15)

(b) locating an instance of the requested native resource in the user isolation scope on behalf of a first user; (virtual device inside of VMM, paragraph [0022], lines 18-21)

and (c) responding to the request for the native resource using the instance of the required native resource located in the user isolation scope. (virtual devices virtualize functionalities of physical devices, paragraph [0026], lines 1-3)

As per claim 2, Kagi teaches

step (b) comprises failing to locate an instance of the requested native resource in the user isolation scope. (paragraph [0063], lines 3-5)

As per claim 3, Kagi teaches

step (c) comprises redirecting the request to the application isolation scope. (paragraph [0028], lines 1-5)

As per claim 4, Kagi teaches

(d) locating an instance of the requested native resource in the application isolation scope; (paragraph [0025], lines 4-6)  
and responding to the request for the native resource using the instance of the requested native resource located in the application isolation scope. (paragraph [0025], lines 5-8)

As per claim 5, Kagi teaches

step (e) comprises creating an instance of the requested native resource in the user isolation scope that corresponds to the instance of the requested native resource located in the application isolation scope and responding to the request for the native resource using the instance of the requested native resource created in the user isolation scope. (paragraph [0026], lines 8-12)

As per claim 6, Kagi teaches

step (d) comprises failing to locate an instance of the requested native resource in the application isolation scope. (paragraph [0063], lines 3-5)

As per claim 7, Kagi teaches

step (e) comprises responding to the request for the native resource using the system-scoped native resource. (paragraph [0023], lines 1-4)

As per claim 8, Kagi teaches

step (e) comprises: creating an instance of the requested native resource in the user isolation scope that corresponds to the instance of the requested resource located in the system scope and responding to the request for the native resource using the instance of the resource created in the user isolation scope. (paragraph [0019], lines 6-10)

As per claim 9, Kagi teaches

the step of hooking a request for a native resource made by a process executing on behalf of a first user. (paragraph [0024], lines 2-5)

As per claim 10, Kagi teaches

the step of intercepting a request for a native resource executing on behalf of a first user. (paragraph [0025], lines 4-7)

As per claim 11, Kagi teaches

the step of intercepting by a file system filter driver a request for a file system native resource executing on behalf of a first user. (paragraph [0026], lines 10-14)

As per claim 12, Kagi teaches

step (a) comprises redirecting to an isolation environment comprising a user isolation scope and an application isolation scope a request for a file made by a process executing on behalf of a first user. (paragraph [0027], lines 3-7)

As per claim 13, Kagi teaches

step (a) comprises redirecting to an isolation environment comprising a user isolation scope and an application isolation scope a request for a registry database entry made by a process executing on behalf of a first user. (paragraph [0026], lines 10-15)

As per claim 14, Kagi teaches

(d) redirecting to the isolation environment a request for the native resource made by a second process executing on behalf of a second user; (paragraph [0025], lines 8-12)

(e) locating an instance of the requested native resource in a second user isolation scope; (paragraph [0025], lines 10-14)

(f) and responding to the request for the native resource using the instance of the native resource located in the second user isolation scope. (paragraph [0025], lines 10-16)

As per claim 15, Kagi teaches

the process executes concurrently on behalf of a first user and a second user.  
(paragraph [0022], lines 4-10)

As per claim 16, Kagi teaches

step (e) comprises failing to locate an instance of the requested native resource in the second user isolation scope. (paragraph [0063], lines 3-5)

As per claim 17, Kagi teaches

step (f) comprises redirecting the request to the application isolation scope.  
(paragraph [0028], lines 2-5)

As per claim 18, Kagi teaches

(d) locating an instance of the requested resource in the application isolation scope; (paragraph [0025], lines 2-5)  
and (e) responding to the request for the native resource using the version of the native resource located in the application isolation scope. (paragraph [0025], lines 3-6)

As per claim 19, Kagi teaches

(d) redirecting to the isolation environment a request for a native resource made by a second process executing on behalf of a first user; (paragraph [0025], lines 8-12)  
(e) locating an instance of the requested native resource in the user isolation scope; (paragraph [0025], lines 10-14)

and (f) responding to the request for the native resource using the instance of the resource located in the user isolation scope. (paragraph [0025], lines 10-16)

As per claim 20, Kagi teaches

step (e) comprises failing to locate an instance of the requested native resource in the user isolation scope. (paragraph [0063], lines 3-5)

As per claim 21, Kagi teaches

step (f) comprises redirecting the request to a second application isolation scope. (paragraph [0025], lines 8-12)

As per claim 22, Kagi teaches

(d) locating an instance of the requested resource in the second application isolation scope; (paragraph [0025], lines 8-12)  
and (e) responding to the request for the native resource using the instance of the native resource located in the second application isolation scope. (paragraph [0025], lines 10-14)

As per claim 23, Kagi teaches

An isolation environment for isolating access by application programs to native resources provided by an operating system, the isolation environment comprising: (see abstract and background)

a user isolation scope storing an instance of a native resource, the user isolation scope corresponding to a user; (virtual machine which performs isolation by virtualizing resources, paragraph [0019], lines 5-15)

and a redirector intercepting a request for the native resource made by a process executing on behalf of the user and redirecting the request to the user isolation scope. (virtual devices virtualize functionalities of physical devices, paragraph [0026], lines 1-3)

As per claim 24, Kagi teaches

the isolation environment further comprises an application isolation scope storing an instance of the native resource. (paragraph [0026], lines 2-6)

As per claim 25, Kagi teaches

the isolation environment further comprises a second application isolation scope storing an instance of the native resource. (paragraph [0025], lines 6-12)

As per claim 26, Kagi teaches

the redirector returns a handle to the requesting process that identifies the native resource. (paragraph [0028], lines 10-14)

As per claim 27, Kagi teaches

a rules engine specifying behavior for the redirector when redirecting the request. (paragraph [0032], lines 4-10)

As per claim 28, Kagi teaches

the redirector comprises a file system filter driver. (paragraph [0032], lines 2-5)

As per claim 29, Kagi teaches

the redirector comprises a function hooking mechanism. (paragraph [0038], lines 4-8)

As per claim 30, Kagi teaches

the function hooking apparatus intercepts an operation selected from the group of file system operations, registry operations, operating system services, packing and installation services, named object operations, window operations, file-type association operations and Component Object Model (COM) server operations. (paragraph [0026], lines 8-15)

As per claim 31, Kagi teaches

the application isolation environment further comprises a second user isolation scope storing a second instance of the native resource. (paragraph [0025], lines 8-12)

As per claim 32, Kagi teaches

the application isolation environment further comprises a second user isolation scope storing an instance of the native resource, the second user isolation scope corresponding to a second user. (paragraph [0025], lines 10-14)

***Response to Arguments***

8. Applicant's arguments with respect to claims 1-32 have been considered but are moot in view of the new ground(s) of rejection.

***Conclusion***

9. The prior art made of record, listed on form PTO-892, and not relied upon is considered pertinent to applicant's disclosure.

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Jay A. Morrison whose telephone number is (571) 272-7112. The examiner can normally be reached on M-F 8-4:30.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Tim Vo can be reached on (571) 272-3642. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see <http://pair-direct.uspto.gov>. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

Jay Morrison  
TC2100

Tim Vo  
TC2100

/Tim T. Vo/  
Supervisory Patent Examiner, Art Unit 2168

Substitute for form 1449/PTO				<b><i>Complete if Known</i></b>	
				Application Number	10/711,737
				Filing Date	September 30, 2004
				First Named Inventor	Lee Laborczfalvi
				Art Unit	2168
				Examiner Name	Jay A. Morrison
Sheet	1	of	3	Attorney Docket Number	2006579-0141

<b>U.S. PATENT DOCUMENTS</b>					
Examiner Initials*	Cite No. <sup>1</sup>	Document Number	Publication Date MM-DD-YYYY	Name of Patentee or Applicant of Cited Document	Pages, Columns, Lines, Where Relevant Passages or Relevant Figures Appear
		Number-Kind Code <sup>2</sup> (if known)			
/JM/	A71*	US-4,253,145	02-24-1981	Goldberg	
/JM/	A72*	US-5,915,085	06-22-1999	Koved	
/JM/	A73*	US-6,453,334	09-17-2002	Vinson et al.	
/JM/	A74*	US-6,735,601	05-11-2004	Subrahmanyam	
/JM/	A75*	US-6,928,469	08-09-2005	Duursma et al.	
/JM/	A76*				
/JM/	A77*	US-6,938,096	08-30-2005	Greschler et al.	
/JM/	A78*	US-7,203,941	04-10-2007	Demsey et al.	
/JM/	A79*	US-7,206,819	04-17-2007	Schmidt	
/JM/	A80*	US-20020083183	06-27-2002	Pujare et al.	
/JM/	A81*	US-20030009538	01-09-2003	Shah et al.	
/JM/	A82*	US-20030101292	05-29-2003	Fisher et al.	
/JM/	A83*	US-20040128250	07-01-2004	Fox et al.	
/JM/	A84*	US-20050065937	03-24-2005	Degenaro et al.	
/JM/	A85*	US-20050193139	09-01-2005	Vinson et al.	

<b>FOREIGN PATENT DOCUMENTS</b>					
Examiner Initials*	Cite No. <sup>1</sup>	Foreign Patent Document	Publication Date MM-DD-YYYY	Name of Patentee or Applicant of Cited Document	Pages, Columns, Lines, Where Relevant Passages Or Relevant Figures Appear
		Country Code <sup>3</sup> -Number <sup>4</sup> -Kind Code <sup>5</sup> (if known)			
/JM/	B86	WO-00/45262	08-03-2000	Sun Microsystems Inc	
/JM/	B87	EP-1037147	09-20-2000	British Telecommunications	
/JM/	B88	WO-01/25894	04-12-2001	Ejasent Inc et al.	
/JM/	B89	WO-01/55813	08-02-2001	Novell Inc	
/JM/	B90	WO-01/95094	12-13-2001	Sun Microsystems Inc.	
/JM/	B91	WO-02/27556	04-04-2002	Appstream Inc	
/JM/	B92	WO-02/27492	04-04-2002	Appstream Inc	
/JM/	B93	EP-1241575	09-18-2002	Hewlett Packard Co	
/JM/	B94	WO-03/029977	04-10-2003	Citrix Systems Inc	

Examiner Signature	/Jay Morrison/	Date Considered	05/12/2008
--------------------	----------------	-----------------	------------

\*EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant. \* CITE NO.: Those application(s) which are marked with an single asterisk (\*) next to the Cite No. are not supplied (under 37 CFR 1.98(a)(2)(iii)) because that application was filed after June 30, 2003 or is available in the IFW. <sup>1</sup> Applicant's unique citation designation number (optional). <sup>2</sup> See Kinds Codes of USPTO Patent Documents at [www.uspto.gov](http://www.uspto.gov) or MPEP 901.04. <sup>3</sup> Enter Office that issued the document, by the two-letter code (WIPO Standard ST.3). <sup>4</sup> For Japanese patent documents, the indication of the year of the reign of the Emperor must precede the serial number of the patent document. <sup>5</sup> Kind of document by the appropriate symbols as indicated on the document under WIPO Standard ST.16 if possible. <sup>6</sup> Applicant is to place a check mark here if English language Translation is attached.

<b>NON PATENT LITERATURE DOCUMENTS</b>			
Examiner Initials	Cite No. <sup>1</sup>	Include name of the author (in CAPITAL LETTERS), title of the article (when appropriate), title of the item (book, magazine, journal, serial, symposium, catalog, etc.), date, page(s), volume-issue number(s), publisher, city and/or country where published.	T <sup>2</sup>
/JM/	C15	"Deploying Internet Information Services (IIS) 6.0", Microsoft Windows Server 2003 Development Kit, page 69, line 25 - page 71, line 8.	

Substitute for form 1449/PTO				<i>Complete if Known</i>	
				Application Number	10/711,737
				Filing Date	September 30, 2004
				First Named Inventor	Lee Laborczfalvi
				Art Unit	2168
				Examiner Name	Jay A. Morrison
Sheet	2	of	3	Attorney Docket Number	2006579-0141

/JM/	C16	"Taking Command of Windows XP", May 3, 2004. URL: <a href="http://media.wiley.com/product_data/excerpt/34/07645582/0764558234.pdf">http://media.wiley.com/product_data/excerpt/34/07645582/0764558234.pdf</a>	
/JM/	C17	Bouras, et al., "Application on Demand System over the Internet", Journal of Network and Computer Applications, Academic Press, New York, NY, US, 28(3):209-232, 2005.	
/JM/	C18	Boyd, et al., "Process Migration: A Generalized Approach Using a Virtualizing Operating System", Proceedings of the 22nd International Conference on Distributed Computing Systems, ICDS 2002, Vienna, Austria, July 2-5, 2002, International Conference on Distributed Computing Systems, Los Alamitos, CA: IEEE Comp. Soc, US, vol. Conf. 22, July 2, 2002, pages 348-355.	
/JM/	C19	Cowan, et al., "Subdomain: Parsimonious Server Security", Proceeding of the 14th Usenix Systems Administration Conference, pages 1-20, 2000. URL: <a href="http://coblitz.codeen.org:3125/citeseer.ist.psu.edu/cache/papers/cs/20700/http:zSzzSzwww.cs.eogi.eduzSz~crispinzSzsubdomain.pdf/cowan00subdomain.pdf">http://coblitz.codeen.org:3125/citeseer.ist.psu.edu/cache/papers/cs/20700/http:zSzzSzwww.cs.eogi.eduzSz~crispinzSzsubdomain.pdf/cowan00subdomain.pdf</a>	
/JM/	C20	Declaration of Non-Establishment of International Search Report, PCT/2005/034178, mailed on January 2, 2006.	
/JM/	C21	European Search Report, 4319MNMs, dated September 25, 2007 in Application No.: 07112164.4-2211	
/JM/	C22	European Search Report, 4324MNMs, dated on January 10, 2008 in Application No.: 07113091.8-2211.	
/JM/	C23	European Search Report, 4325MNMs, dated on January 15, 2008 in Application No.: 07113105.6-2211.	
/JM/	C24	International Search Report, PCT/US2005/033994, mailed on February 15, 2006.	
/JM/	C25	International Search Report, PCT/US2005/034177, mailed on February 21, 2006.	
/JM/	C26	International Search Report, PCT/US2005/034302, mailed on May 8, 2006.	
/JM/	C27	International Search Report, PCT/US2005/034449, mailed on March 3, 2006.	
/JM/	C28	International Search Report, PCT/US2006/037602, mailed on May 31, 2007.	
/JM/	C29	Pietrek, "Intercepting API Functions in Win32", PC Magazine - US Edition, Ziff-Davis, New York, NY, US, 13(19):307-312, 11 August 1994.	
/JM/	C30	Sampemane, et al., "Access control for active spaces", Computer Security Applications Conference, 2002. Proceedings. 18th Annual 9-13 Dec. 2002, Piscataway, NJ, USA, IEEE, pages 343-352.	
/JM/	C31	Souder, et al., "A tool for securely integrating legacy systems into a distributed environment", Reverse Engineering, 1999, Proceedings. Sixth Working Conference on Atlanta, GA, USA 6-8 Oct. 1999, Los Alamitos, CA, USA, IEEE Comput. Soc., US, pages 47-55.	
/JM/	C32	U.S. Office Action, dated November 14, 2007, in U.S. Application No.:10/711,734.	
/JM/	C33	Written Opinion of International Searching Authority, PCT/US2005/034178, mailed on January 2, 2006.	
/JM/	C34	Written Opinion of International Searching Authority, PCT/US2005/034302, mailed on May 8, 2006.	
/JM/	C35	Written Opinion of International Searching Authority, PCT/US2005/034177, mailed on February 21, 2006.	
/JM/	C36	Written Opinion of International Searching Authority, PCT/US2006/037602, mailed on May 31, 2007.	
/JM/	C37	Written Opinion of the International Searching Authority, PCT/US2005/034449, mailed on March 3, 2006.	
/JM/	C38	Written Opinion of the International Searching Authority, PCT/US2005/033994, mailed on February 15, 2006.	
/JM/	C39	Zhenkai, et al., "Isolated Program Execution: An Application Transparent Approach for	

Substitute for form 1449/PTO				<i>Complete if Known</i>	
				Application Number	10/711,737
				Filing Date	September 30, 2004
				First Named Inventor	Lee Laborczfalvi
				Art Unit	2168
				Examiner Name	Jay A. Morrison
Sheet	3	of	3	Attorney Docket Number	2006579-0141

/JM/	Executing Untrusted Programs", Computer Security Applications Conference, 2003. Proceedings. 19th Annual 8-12 Dec. 2003, Piscataway, NJ, USA, IEEE, 8 December 2003, pages 182-191.	
------	---	--

Examiner Signature	/Jay Morrison/	Date Considered	05/12/2008
--------------------	----------------	-----------------	------------

\*EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

<sup>1</sup>Applicant's unique citation designation number (optional). <sup>2</sup>Applicant is to place a check mark here if English language Translation is attached.

<b>Notice of References Cited</b>		Application/Control No.	Applicant(s)/Patent Under Reexamination LABORCZFALVI ET AL.	
		10/711,737	Examiner JAY A. MORRISON	Art Unit 2168      Page 1 of 1

**U.S. PATENT DOCUMENTS**

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
*	A	US-5,819,044 A	10-1998	Kawabe et al.	709/226
*	B	US-2005/0091658 A1	04-2005	Kavalam et al.	718/104
*	C	US-2005/0091214 A1	04-2005	Probert et al.	707/009
*	D	US-2006/0064697 A1	03-2006	Kagi et al.	718/103
*	E	US-7,103,625 B1	09-2006	Hipp et al.	709/201
*	F	US-2007/0067366 A1	03-2007	Landis, John A.	707/205
	G	US-			
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

**FOREIGN PATENT DOCUMENTS**

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

**NON-PATENT DOCUMENTS**

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Boyd et al., "Preemptive module replacement using the virtualizing operating system realizing multi-dimensional software adaptation," Proceedings of the ACM Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), in conjunction with the 16th Annual ACM International Conference on Supercomputing, (New York City, NY), June 2002.
	V	Boyd et al., "Virtualizing Operating Systems for Seamless Distributed Environments", in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, vol. 2, November 2000, pp. 735-740.
	W	Potter et al, "Secure Isolation and Migration of Untrusted Legacy Applications", Computer Science Department, Columbia University, Columbia University Technical Report CU-CS-005-04, January 2004.
	X	

\*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)  
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

# Secure Isolation and Migration of Untrusted Legacy Applications

Shaya Potter Jason Nieh Dinesh Subhraveti

*Computer Science Department  
Columbia University*

{spotter, nieh, dinesh}@cs.columbia.edu

*Columbia University Technical Report CUCS-005-04, January 2004*

## Abstract

Existing applications often contain security holes that are not patched until after the system has already been compromised. Even when software updates are applied to address security issues, they often result in system services being unavailable for some time. To address these system security and availability issues, we have developed peas and pods. A pea provides a least privilege environment that can restrict processes to the minimal subset of system resources needed to run. This mechanism enables the creation of environments for privileged program execution that can help with intrusion prevention and containment. A pod provides a group of processes and associated users with a consistent, machine-independent virtualized environment. Pods are coupled with a novel checkpoint-restart mechanism which allows processes to be migrated across minor operating system kernel versions with different security patches. This mechanism allows system administrators the flexibility to patch their operating systems immediately without worrying over potential loss of data or needing to schedule system downtime. We have implemented peas and pods in Linux without requiring any application or operating system kernel changes. Our measurements on real world desktop and server applications demonstrate that peas and pods impose little overhead and enable secure isolation and migration of untrusted applications.

## 1 Introduction

As software complexity grows and computers become more interconnected, the need for effective computer security increases. Complex software often contains programming errors, some of which may lead to vulnerabilities that can be exploited by attackers who gain access to those applications. Standard security models employed by commodity operating systems, such as Unix, do not help this situation. Because Unix lumps all privileges together as root, an application that only periodically needs one privilege still needs to run as root, providing it with all privileges. An attacker can thus gain root privileges by exploit-

ing a weakness in an application run as root. Consequently, Internet accessible services offer prime opportunities for remote attackers to gain access to applications running with privilege.

Security problems can wreak havoc on an organization's computing infrastructure. To prevent this, software vendors frequently release patches that can be applied to address security issues that have been discovered. However, software patches need to be applied to be effective. It is not uncommon for systems to continue running unpatched applications long after a security exploit has become well-known [35]. This is especially true of the growing number of server appliances intended for very low-maintenance operation by less skilled users. Furthermore, once a patch has been released, exploits of unpatched applications based on reverse engineering the patch now occur as quickly as a month later whereas such exploits took closer to a year just a couple years ago [23].

Software updates to existing applications may not address security problems that result from users accidentally downloading and executing malicious code. Recently a security hole was discovered in a popular mp3 player [19] that could result in arbitrary code being executed if a user played a maliciously constructed mp3. If the mp3 player were run within a simple sandbox that limited the player to one's collection of mp3s, the damage the malicious code could accomplish would be severely limited. Over the years, complex services like Sendmail have similarly been exploited to allow malicious code to be run within its context. Since Sendmail runs with privilege, the malicious code also runs with privilege. A sandbox can be used to protect an entire machine from a faulty service, such as Sendmail. However, these services don't run by themselves, but also depend on other aspects of the machine, such as programs a user might want to call from a Procmail script to filter their mail. Consequently, one might end up including the entire machine within the sandbox. Since common sandboxes simply provides a single namespace, they don't provide good security solutions for the complex services in use today.

Furthermore, even when software updates are applied to address security issues, they commonly result in system services being unavailable. Patching an operating system can result in the entire system having to be down for some period of time. If a system administrator chooses to fix an operating system security problem immediately, he risks upsetting his users because of loss of data. Therefore, a system administrator must schedule downtime in advance and in cooperation with all the users, leaving the computer vulnerable until repaired. If the operating system is patched successfully, the system downtime may be limited to just a few minutes during the reboot. If the patch is not successful, downtime can extend for many hours while the problem is diagnosed and a solution is found. For systems that need to provide a high degree of availability, downtime due to security-related issues is not only inconvenient but costly as well. While application servers can sometimes mirror application state between servers and allow an application to continue even when one server has to be taken down, they only work in specific situations. For instance, a regular user's desktop can not be mirrored between servers. Even for applications that can mirror their data, the application has to be designed to interface with the mirroring architecture, resulting in application specific solutions that are difficult to generalize.

We introduce Pea-Pods to provide a solution to these security problems. Pea-Pods provide two key abstractions, peas (Protection and Encapsulation Abstraction) and pods (PrOcess Domain). A pod is a lightweight migratable virtual execution environment that looks just like the underlying operating system environment. A pea is a least privilege environment within a pod that allows access to a subset of processes and resources in the pod. In tandem, peas and pods decouple process execution from the underlying operating system to provide transparent, secure isolation and migration of untrusted applications. Pea-Pods can isolate untrusted applications within sandboxes, preventing them from causing harm to the underlying system or other applications if they are compromised.

Pea-Pods can encapsulate a group of processes within a migratable sandbox environment that can be transparently moved from one machine to another, even when the systems are running different operating system versions with different security and maintenance patches. This enables security patches to be applied to operating systems in a timely manner with minimal impact on the availability of application services by migrating applications to another machine that has already been updated while the original system is brought down for security upgrades and maintenance. Once the original machine has been updated, applications can be migrated back and continue to execute even though the underlying operating system has changed. Pea-Pods provide migration using a checkpoint-restart mechanism that can also enable application services to be check-pointed before a system goes down and restarted when it

comes back up. This provides fast recovery from system downtime even when other machines are not available to migrate application services, as well as providing a general solution that any application can take advantage of.

Pea-Pods achieve these goals through three distinguishing characteristics. First, a pod provides a consistent private virtual namespace that gives all processes within it the same virtualized view of the system. This virtualized view isolates sandboxed processes from the underlying system by associating virtual identifiers with operating system resources and only allowing access to resources that are made available within the virtualized namespace. This isolation mechanism provides a simple way to control what operating system resources are accessible to a group of processes. Similarly, it allows a pod to define a complete set of users which can be distinct from those supported by the underlying system.

Second, a pea provides a least privilege encapsulation layer within a pod that can limit certain processes from interacting with other processes and accessing file system and network resources. This is effective for preventing compromised applications from attacking other processes and resources of the system. We provide intuitive tools to easily and dynamically create Pea-Pods tailored for individual applications or groups of applications.

Third, Pea-Pod virtualization is integrated with a checkpoint-restart mechanism that decouples processes from dependencies on the underlying system and maintains process state semantics to enable processes to be migrated across different machines. The checkpoint-restart mechanism employs an intermediate format for saving the state associated with processes and Pea-Pod virtualization. This format provides a high degree of portability to support process migration across machines that are running operating systems that differ in the security and maintenance patches applied. It also enables application services to be check-pointed on a system and restarted after the underlying operating system is upgraded and the system is restarted.

We have implemented Pea-Pods in a prototype system as a loadable Linux kernel module. We have used this prototype to securely isolate and migrate a wide range of unmodified legacy and network applications. We measure the performance and demonstrate the utility of Pea-Pods across multiple systems running different Linux 2.4 kernel versions using three real-world application scenarios, including a full KDE desktop environment with a suite of desktop applications, an Apache/MySQL web server and database server environment, and a Sendmail/Procmail e-mail processing environment. Our performance results show that Pea-Pods can provide secure isolation and migration functionality on real world applications with low overhead.

This paper describes how Pea-Pods can isolate applications to limit their ability to attack a system and how Pea-Pods can migrate applications across operating system kernel changes to facilitate kernel maintenance and secu-

riety updates with minimal application downtime. Section 2 describes the pea and pod abstractions in further detail. Section 3 presents the virtualization architecture to support the Pea-Pod model. Section 4 discusses the Pea-Pod checkpoint-restart mechanisms used to facilitate migration across operating system kernels that may differ in maintenance and security updates. Section 5 analyzes the security of Pea-Pods and illustrates the utility of the system in several application scenarios. Section 6 presents experimental results evaluating the overhead associated with Pea-Pods and measures the system performance in providing secure isolation and migration for several application scenarios. Section 7 discusses related work. Finally, we present some concluding remarks.

## 2 Pea-Pod Model

The Pea-Pod model provides two key abstractions, pods (PrOcess Domain) and peas (Protection and Encapsulation Abstraction). Pods enable secure isolation and migration of application components that only need to interact via the file system or Internet communication. Peas provide fine-grain isolation among application components that may need to interact using interprocess communication mechanisms, including signals, shared memory, IPC messages and semaphores, and process forking and execution.

A pod is a host-independent virtualized view of an operating system in which a group of processes can be executed. A pod may contain one or many processes, and a system may contain one or many pods. The pod abstraction provides the same application interface as the underlying operating system so that legacy applications can execute in the context of a pod without any modification. Processes within a pod can make use of all available operating system services, just like processes executing in a traditional operating system environment. Unlike a traditional operating system, the pod abstraction provides a self-contained unit that can be isolated from the system, checkpointed to secondary storage, migrated to another machine, and transparently restarted, as shown in Figure 1. This is made possible because each pod has its own private, virtual namespace. All operating system resources are only accessible to processes within a pod through the pod's private, virtual namespace.

A pod namespace is private in that only processes within the pod can see the namespace. It is private in that it masks out resources that are not contained within the pod, including processes outside of the pod. Processes inside a pod appear to one another as normal processes that can communicate using traditional IPC mechanisms. Other processes outside a pod do not appear in the namespace and are therefore not able to interact with processes inside a pod using IPC mechanisms such as shared memory and signals. Instead, processes outside the pod can only interact with processes inside the pod using network communication and

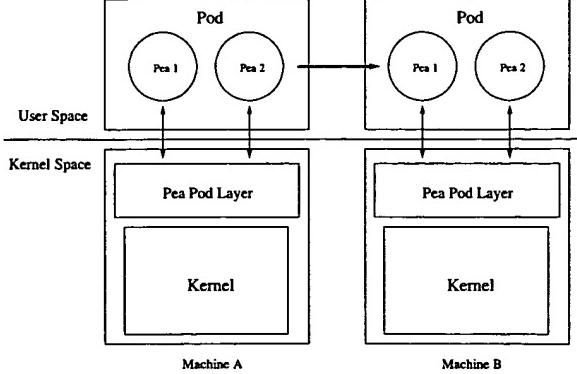


Figure 1: Pea-Pod migration

shared files that are normally used to support process communication across machines.

A pod namespace is virtual in that all operating system resources including processes, user information, files, and devices are accessed through virtual identifiers within a pod. These virtual identifiers are distinct from host-dependent resource identifiers used by the operating system. The pod virtual namespace provides a host-independent view of the system by using virtual identifiers that remain consistent throughout the life of a process in the pod, regardless of whether the pod moves from one system to another. Since the pod namespace is separate from the underlying operating system namespace, the pod namespace can preserve this naming consistency for its processes even if the underlying operating system namespace changes, as may be the case in migrating processes from one machine to another.

The pod private, virtual namespace enables secure isolation of applications by providing complete mediation to operating system resources. Pods can restrict what operating system resources are accessible within a pod by simply not providing identifiers to such resources within its namespace. A pod only needs to provide access to resources that are needed for running those processes within the pod. It does not need to provide access to all resources to support a complete operating system environment. For example, a pod can easily provide a least privilege environment tailored to the needs of an application services. If one had a web server that just served up static content, one could easily setup the pod to only contain the files the web server needs to run as well as the content it wants to serve. If the web server application gets compromised, the pod limits the ability of an attacker to further harm the system since the only resources he has access to are the ones explicitly needed by the service. Since the pod namespace provides the same application interface as the underlying operating system, pods can provide complete mediation without modifying, recompiling, or relinking applications.

The pod private, virtual namespace enables process migration by providing a consistent, host-independent view of the underlying operating system. Operating system re-

source identifiers such as process IDs (PIDs) must remain constant throughout the life of a process to ensure its correct operation. However, when a process is moved from one operating system to another, there is no guarantee that the underlying operating system will provide the same identifiers to a migrated process; those identifiers may in fact already be used by other processes in the system. The pod namespace addresses these issues by providing consistent, virtual resource names in place of host-dependent resource names such as PIDs. Names within a pod are trivially assigned in a unique manner in the same way that traditional operating systems assign names, but such names are localized to the pod. Since the namespace is private to a given pod, there are no resource naming conflicts for processes in different pods. There is no need for the pod namespace to change when the pod is migrated, which allows pods to ensure that identifiers remain constant throughout the life of the process, as required by legacy applications that use such identifiers.

A process can run inside a pod, but there are times when it is desirable to further restrict a process inside a pod in terms of the pod resources it can access. For example, in a conventional e-mail system, one will have a privileged SMTP daemon, such as Sendmail, and a non-privileged delivery agent, such as Procmail. While the Sendmail server runs with privilege, it actually needs a very small resource namespace. However, the Procmail delivery agent can make use of programs, such as SpamAssassin, to enable users to filter their e-mail effectively. Since these two programs need to interact directly, they can not be run in separate pods. Peas are introduced for the purpose of allowing these programs to interact, while restricting them to smaller resource namespaces. A pea is an abstraction that can contain a subset of processes within a pod and restrict those processes to accessing only a subset of pod resources. Pods can contain a group of processes, but the group may be composed of interacting components with different resource needs. Peas can separate these components within the pod by providing fine-grained and dynamic resources restrictions on differing sets of processes. The pea abstraction allows for processes running within a pod to have varying levels of isolation among them by running them in separate peas.

A pea achieves isolation levels by controlling what resources of a pod its processes are allowed to access and interact with. Peas provide a “see, but don’t touch” resource restriction model. For example, a process in a pea may be able to see file system resources and processes available to other peas, but can be restricted from accessing them. Unlike processes in separate pods, processes in separate peas in a single pod can “see each other” in that they share the same namespace and can be allowed to interact using traditional interprocess communication mechanisms. Processes can also be allowed to move from one pea to another in the same pod. However, by default processes in separate peas

“can’t touch” any resource outside of its pea, be it a process pid or file system entry. Peas can support a wide range of resource restriction policies. By default, processes contained in a pea can only interact with other processes in the same pea. They have no access to other resources, such as file system and network resources or processes outside of the pea. This provides for a set of fail safe defaults, as any extra access has to be explicitly allowed by the administrator.

Many peas can be running side by side to provide flexibility in implementing a least privilege system for programs that are composed of multiple components that must work together, but do not all need the same level of privilege. One usage scenario would be to have a severely resource limited pea in which a privileged process executes but allowing the process to use traditional Unix semantics to work with less privileged programs that are in less resource restricted peas. One use of this is the mail delivery services already described, one can create two separate peas for Sendmail and Procmail to run within. It can similarly be used to allow a web server the ability to serve dynamic content via CGI in a more secure manner. Since the web server and the CGI scripts need separate levels of privilege, as well as different resource requirements, they shouldn’t have to run within the same security context. By configuring two separate peas for a web service, one for the web server to run within, and a separate for the specific CGI programs it wants to execute, one limits the damage that can occur if a fault is discovered within the web server. If one manages to execute malicious code within the context of the web server, one can only make use of resources that are allocated to the web server’s pea, as well as only execute the specific programs that are needed as CGIs. Since the CGI programs will also only run within their specific security context, the ability for malicious code to do harm is severely limited.

Peas and pods together provide secure isolation based on flexible resource restriction for programs as opposed to restricting access based on users. Pea-Pods also do not subvert underlying system restrictions based on user permissions, but instead complement such models by offering additional resource control based on the environment in which a program is executed. Instead of allowing programs with root privileges to do anything they want to a system, Pea-Pods enable a system to control the execution of such programs to limit their ability to harm a system even if they are compromised. Pea-Pods provide program-based resource restriction for file access, device access, network access, root privileges, process interactions, process transitions among peas, and resource utilization. Pea-Pods can restrict root privileges by disallowing certain operating system services for a given pea or pod. Pea-Pods can restrict process interactions by disallowing interprocess communication with processes outside of a pod, and by limiting such interactions among processes in separate peas

in a pod. Pea-Pods can dynamically control the ability of processes to transition between peas, enabling processes to have different dynamic privileges during their execution. Pea-Pods can control the resources that processes consume in a pea or pod to limit denial of service attacks against the system. Due to space constraints, the Pea-Pod resource usage model is not discussed further in this paper.

### 3 Pea-Pod Virtualization

To support the Pea-Pod abstraction design of secure and isolated namespaces on commodity operating systems, we employ a virtualization architecture that operates between applications and the operating system, without requiring any changes to applications or the operating system kernel. This virtualization layer is used to translate between the Pea-Pod namespaces and the underlying host operating system namespace. It also protects the host operating system from dangerous privileged operations that might be performed by processes within the Pea-Pod, as well as protecting those processes from processes outside of the Pea-Pod on the host. Pea-Pod virtualization is used to provide isolation of peas and pods as well as enable pods to be migratable. The virtualization support for pod migration is based on Zap [28].

#### 3.1 Pod Virtualization

Pods are supported using virtualization mechanisms that translate between pod virtual resource identifiers and operating system resource identifiers. Every resource that a process in a pod accesses is through a *virtual name* which corresponds to an operating system resource identified by a *physical name*. When an operating system resource is created for a process in a pod, such as with process or IPC key creation, instead of returning the corresponding physical name to the process, the pod virtualization layer catches the physical name value, and returns a private virtual name to the process. Similarly, any time a process passes a virtual name to the operating system, the virtualization layer catches it and replaces it with the appropriate physical name. The key pod virtualization mechanisms used are a system call interposition mechanism and the `chroot` utility with file system stacking for file system resources.

Pod virtualization employs system call interposition to wrap existing system calls to check and replace arguments that take virtual names with the corresponding physical names before calling the underlying original system call. Similarly, the wrapper is used to capture physical name identifiers that the original system calls return and return corresponding virtual names to the calling process running inside the pod. Pod virtual names are maintained consistently as a pod migrates from one machine to another and are remapped appropriately to underlying physical names

that may change as a result of migration. Pod system call interposition also masks out processes inside of a pod from processes outside of a pod to remove any interprocess host dependencies across pod boundaries. System call interposition is used to virtualize operating system resources including process identifiers, keys and identifiers for IPC mechanisms such as semaphores, shared memory, and message queues, and network addresses.

Pod virtualization uses system call interposition to determine the network accessibility of pod processes. Pods provide the same semantic interface to applications as regular machines, which provide Internet accessible and local-host addresses. Therefore, pods also provide two types of networking addresses. Pods provide one that is only accessible to processes in a pod and one that is accessible on the Internet. A pod restricts its processes to the set of network addresses given to the pod by using the same virtual to physical mapping concepts of PID and IPCs. Processes within a pod make use of a virtual name for a network address. Since the regular pod virtualization rules take effect, processes are confined to the appropriate addresses.

Pod virtualization employs the `chroot` utility and file systems stacking to provide each pod with its own file system namespace that can be separate from the regular host file system. The pod file system can be composed from loopback mounts from the host for pods that are only checkpointed and restarted on the same machine. Similarly, one can make use of a portable hard drive that one moves between the different hosts one wants to migrate within. More commonly, the pod file system is composed from remote mounts via a network file system such as NFS so that the same files can be made consistently available as a pod is migrated from one machine to another. More specifically, when a pod is created or moved to a host, a private directory named according to a pod identifier is created on the host to serve as a staging area for the pod's virtual file system. Within this directory, the various network-accessible directories that the pod is configured to access will be mounted from a network file server. For example, from a Unix-centric viewpoint, this set of directories could include `/etc`, `/lib`, `/bin`, `/usr`, and `/tmp`. The `chroot` system call is then used to set the staging area as the root directory for the pod, thereby achieving file system virtualization with negligible performance overhead. This method of file system virtualization provides an easy way to restrict access to files and devices from within a pod. This can be done by simply not including file hierarchies and devices within the pod file system namespace. If files and devices are not mounted within the pod virtual file system, they are not accessible to pod processes.

Because commodity operating systems are not built to support multiple namespaces, a security issue that pod virtualization must address is that there are many ways to break out of a standard chrooted environment, especially if one allows the `chroot` system call to be used by pro-

cesses in a pod. Pod file system virtualization enforces the chrooted environment and ensures that the pod's file system is only accessible to processes within the given pod by using a simple form of file system stacking to implement a barrier. File systems provide a permission function that determines if a process can access a file. For example, if a process tries to access a file a few directories below the current directory, the permission function is called on each directory as well as the file itself in order. If any of calls determine that the process doesn't have permission on a directory, the chain of calls end. Even, if the permission function determines that the process would have access to the file itself, it must have permission to walk the directory hierarchy to the file to access it. We implement a barrier by simply stacking a small pod-aware file system on top of the staging directory that overloads the underlying permission function to prevent processes running within the pod from accessing the parent directory of the staging directory, and to prevent processes running only on the host from accessing the staging directory. This effectively confines a process in a pod to the pod's file system by preventing it from ever walking past the pod's file system root.

While any network file system can be used with pods to support migration, we focus on NFS because it is the most commonly used network file system. Pods can take advantage of the user identifier (UID) security model in NFS to support multiple security domains on the same system running on the same operating system kernel. For example, since each pod can have its own private file system, each pod can have its own `/etc/passwd` file that determines its list of users and their corresponding UIDs. In NFS, the UID of a process determines what permissions it has in accessing a file. By default, pod virtualization keeps process UIDs consistent across migration and keeps process UIDs the same in the pod and operating system namespaces. However, since the pod file system is separate from the host file system, a process running in the pod is effectively running in a separate security domain from another process with the same UID that is running directly on the host system. Although both processes have the same UID, each process is only allowed to access files in its own file system namespace. Similarly, multiple pods can have processes running on the same system with the same UID, but each pod effectively provides a separate security domain since the pod file systems are separate from one another.

The pod UID model supports an easy-to-use migration model when a user may be working in one administrative domain and then moves to another. Even if the user has computer accounts in both administrative domains, it is unlikely that the user will have the same UID in both domains if they are administratively separate. Nevertheless, pods can enable the user to run the same pod with access to the same files in both domains. Suppose the user has UID 100 on a machine in administrative domain A and starts a pod connecting to a file server residing in domain A. Sup-

pose that all pod processes are then running with UID 100. When the user moves to a machine in administrative domain B where he has UID 200, he can migrate his pod to the new machine and continue running processes in the pod. Those processes can continue to run as UID 100 and continue to access the same set of files on the pod file server, even though the user's real UID has changed. While this example considers the case of having a pod with all processes running with the same UID, it is easy to see that the pod model supports pods that may have running processes with many different UIDs.

Because the root UID 0 is privileged and treated specially by the operating system kernel, pod virtualization also treat UID 0 processes inside of a pod in a special way to prevent them from breaking the pod abstraction, accessing resources outside of the pod, and causing harm to the host system. While a pod can be configured for administrative reasons to allow full privileged access to the underlying system, we focus on the case of pods for running application services which do not need to be used in this manner. Pods do not disallow UID 0 processes, which would limit the range of application services that could be run inside pods. Instead, pods provide restrictions on such processes to ensure that they function correctly inside of pods.

While a process is running in user space, the UID it runs as doesn't have any effect. Its UID only matters when it tries to access the underlying kernel via one of the kernel entry points, namely devices and system calls. Since a pod already provides a virtual file system that includes a virtual `/dev` with a limited set of secure devices, the device entry point is already secured. The only system calls of concern are those that could allow a root process to break the pod abstraction. Only a small number of system calls can be used for this purpose. Pod virtualization classifies these system calls into three classes that need to be protected.

The first class of system calls are those that only affect the host system and serve no purpose within a pod. Examples of these system calls include those that load and unload kernel modules or that reboot the host system. Since these system calls only affect the host, they would break the pod security abstraction by allowing processes within it to make system administrative changes to the host. System calls that are part of this class are therefore made inaccessible by default to processes running within a pod.

The second class of system calls are those that are forced to run unprivileged. Just like NFS, by default, squashes root on a client machine to act as user `nobody`, pod virtualization forces privileged processes to act as the `nobody` user when it wants to make use of some system calls. Examples of these system calls include those that set resource limits and `ioctl` system calls. Since system calls such as `setrlimit` and `nice` can allow a privileged process to increase its resource limits beyond predefined limits imposed on pod processes, privileged processes are by default treated as unprivileged when executing these system calls

within a pod. Similarly, the `ioctl` system call is a system call multiplexer that allows any driver on the host to effectively install its own set of system calls. Since the ability to audit the large set of possible system calls is impossible given that pods may be deployed on a wide range of machine configurations that are not controlled by the Pea-Pod system, pod virtualization conservatively treats access to this system call as unprivileged by default.

The final class of system calls are calls that are required for regular applications to run, but have options that will give the processes access to underlying host resources, breaking the pod abstraction. Since these system calls are required by applications, the pod checks all their options to ensure that they are limited to resources that the pod has access to, making sure they aren't used in a manner that breaks the pod abstraction. For example, the `mknod` system call can be used by privileged processes to make named pipes or files in certain application services. It is therefore desirable to make it available for use within a pod. However, it can also be used to create device nodes that provide access to the underlying host resources. To limit how the system call is used, the pod system call interposition mechanism checks the options of the system call and only allows it to continue if it's not trying to create a device.

## 3.2 Pea Virtualization

Peas are supported using virtualization mechanisms that impose levels of isolation among processes running within a single pod in separate peas by labeling resources and enforcing a simple set of configurable rules. For example, when a process is created in a pea, its process identifier is tagged with the identifier of the pea in which it was created. A process's ability to access pod resources is then dictated by the set of rules associated with its pea. Like pod virtualization, the key pea virtualization mechanisms used are a system call interposition mechanism and file system stacking for file system resources.

Pea virtualization employs system call interposition to wrap existing system calls to enforce restrictions on process interactions by controlling access to process and IPC virtual identifiers. Since each resource is labeled with the pea in which it was created, the system call interposition mechanism simply checks if the pea labels of the calling process and the resource to be touched are the same or different, providing an effective means of enforcing the pea's "see, but don't touch" model. For example, if a process in one pea would try to send a signal to another process in a separate pea by using the `kill` system call, the system would return an error value of `EPERM`, as the process exists, just this process has no permission to signal it. On the other hand, a parent is able to use the `wait` system call to wait on a child process, even if that child process is running within a separate pea since wait doesn't "touch" a process by affecting its execution.

When a new program is executed one might want to switch pea security domains. Peas support a single type of pea specific rule that let a pea determines how a process can transition from one its own pea to another. This rule is specified by a program filename and pea identifier. A pea may have multiple rules of this type. The rule specifies that a process should be moved into the pea specified by the pea identifier if it executes the program specified by the given filename. This is useful when it is known what a process will execute and it is desirable to have that program execution occur in an execution environment with different resource restrictions. For example, an Apache web server running in a pea may want to execute its CGI child processes in a more restrictive pea. This is supported via system call interposition by intercepting the `exec` system call and changing peas if a matching pea transition rule is specified for the pea in which the calling process is executing. Note that pea transition rules are one-way transitions that do not enable a process to return to its previous pea unless its current pea explicitly provides such rules.

System call interposition is also used to control network access for processes inside the pea. Peas provide two networking rules, one to allow processes in the pea to make outgoing network connections on a pod's virtual network adapters, the other to allow processes in the pea to bind to specific ports on the adapter to receive incoming connections. Pea rules can allow complete access to a pod network adapter, or only allow access on a per port basis. Since any network access occurs through system calls, peas simply check the options of the networking system call to ensure that it is allowed to perform the specified action.

Pea virtualization employs a set of file system rules and file systems stacking to provide each pea with its own permission set on top of the pod file system. To provide a least privilege environment, processes shouldn't have access to file system privileges they don't need. For example, while Sendmail has to write to `/var/spool/mqueue`, it only has to read its configuration from `/etc/mail` and should not need to have write permissions on its configuration. To implement such a least privilege environment, peas enable files to be tagged with additional permission rules that overlay the respective underlying file permissions. File system permissions determine access rights based on the user identity of the process while pea file permission rules determine access rights based on the pea context in which a process is executed. Each pea file rule can selectively allow or deny use of the underlying read, write and execute permissions of a file on a per pea basis. The underlying file permission is always enforced, but pea permissions can further restrict whether the underlying permission is allowed to take effect. The final permission is achieved by performing a bit-wise AND operation on both the pea and file system permissions. For example, if the pea permission allowed for read and execute, the permission set of `r-x` would be triplicated to `r-xr-xr-x` for the 3 sets of Unix permissions and

the bitwise AND operation would effectively mask out any write permission that the underlying file system might allow. This prevents any process in the pea from opening the file and modifying it.

Enforcing on disk labeling of every single file is intractable if the underlying file system is going to be used for multiple disparate pods and peas. Since each pea in each pod might make use of similar underlying files but have different permission schemes, storing the pea permission data on disk effectively is not feasible. Instead, peas support the ability to dynamically label each file within a pod’s file system based on two simple path matching rules, *path specific rules* and *directory default rules*. A path specific rule matches an exact path on the file system. For instance, if there’s a path specific rule for `/home/user/file`, only that file will be matched with the appropriate permission set. On the other hand, if there’s a directory default rule for the directory `/home/user/` any file under that directory in the directory tree can match it, and inherit its permission set.

Given a set of path specific and directory default rules for a pea, the algorithm for determining what rule matches to what path starts with the complete path and walks up the path to the root directory until it finds a matching rule. The algorithm can be described in four simple steps:

1. If the specific path has a *path specific rule*, return that rule set.
2. Otherwise, choose the path’s directory as the current directory to test.
3. If the directory being tested has a *directory default rule*, return that rule set.
4. Otherwise set its parent as the current directory to test and go back to step 3.

This ensures that if there’s no *path specific rule*, the closest *directory default rule* to the specified path becomes the rule for that path. Also, since by default peas give the root directory “`/`” a *directory default rule* denying all permissions, the default for every file on the system, unless otherwise specified is deny, ensuring a fail safe default setup.

The semantics of pea file permission rules are based on file path name. If a file has more than one path name, such as via a hard link, both have to be protected by the same rule, otherwise depending on how the underlying file is accessed the permission set it gets will be non-deterministic as the inode cache will contain the permission set of the path name that was opened initially. This is only an issue on setup of a Pea-Pod, as once its setup, any hard links that are created will obey the regular file system rules, which include being unable to hard link to a path one’s pea doesn’t have access to, as well as any new hard link path name that gets created is given a path specific rule equivalent to the original path’s rule.

The pea architecture makes use of the pod’s stackable file system to integrate the pea file system namespace restrictions into the regular kernel permission model. It accomplishes this by stacking on top of the file system’s *lookup* function which fills in the respective file’s inode structure, and the *permission* function which makes use of the stored permission data to make simple permission determinations. Since a file system’s permission function is a standard part of the operating system kernel’s security infrastructure, no changes have to be made to the kernel’s file system security infrastructure.

The stackable file system uses a unique set of hash tables that it organizes in a tree structure to mimic the underlying file system. Every directory can be represented by a hash table, and entries in the hash table correspond to directory entries that have pea file system rules. If a directory entry is an actual directory, it would have a corresponding child hash table. Looking up the appropriate rule for any path name is simply parsing the path name into directory entry tokens, and performing a token by token traversal of the tree of hash tables. This traversal results in finding the rule that best matches the pathname, based on the decision algorithm given above. Since hashing of tokens is fast, one can quickly traverse the tree in  $O(h)$  time, where  $h$  is the height of the file system tree, no matter how many rules the file system enforces. The stackable file system is made even faster by the fact that the rule lookup doesn’t have to be done often, since we store the data in the file system’s inode structure and the kernel caches the inode structure for later use.

## 4 Migration Across Different Kernels

To maintain application service availability without losing important computational state as a result of system downtime due to operating system upgrades, Pea-Pods provide a checkpoint-restart mechanism that allows pods to be migrated across machines running different operating system kernels. Upon completion of the upgrade process, the respective Pea-Pod and its applications are restored on the original machine now with an upgraded operating system. We assume here that the systems have not been compromised and that any kernel security holes on the unpatched system have not yet been exploited on the system; migrating across kernels that have already been compromised is beyond the scope of this paper.

We also limit our focus to migrating between machines with a common CPU architecture with kernel differences that are limited to maintenance and security patches. These patches often correspond to changes in the minor version number of the kernel. For example, the Linux 2.4 kernel has more than twenty minor versions. Even within minor version changes, there can be significant changes in kernel code. Table 1 shows the number of files that have been changed in various subsystems of the Linux 2.4 kernel

Type	.c Files	Changed	Percentage
Drivers	2221	2079	93.6
Arch	2694	2351	87.2
FS	524	488	93.1
Network	422	352	83.4
Core Kernel	27	22	81.4
VM	20	20	100
IPC	4	4	100

Table 1: Kernel Changes within the Linux 2.4 Series

across different minor versions. For example, all of the files for the VM subsystem were changed since extensive modifications were made to implement a completely new page replacement mechanism in Linux. Many of the Linux kernel patches contain security vulnerability fixes, which are typically not separated out from other maintenance patches. We similarly limit our focus to where the application’s execution semantics, such as how threads are implemented and how dynamic linking is done, do not change. On the Linux kernels this is not an issue as all these semantics are enforced by user-space libraries. Whether one uses kernel or user threads, or one how libraries are dynamically linked into a process is all determined by the respective libraries on the file system. Since the Pod has access to the same file system on whatever machine it is running on, these semantics stay the same.

To support migration across different kernels, Pea-Pods use a checkpoint-restart mechanism that employs an intermediate format to represent the state that needs to be saved on checkpoint. On checkpoint, the intermediate format representation is saved and digitally signed to enable the restart process to verify the integrity of the image. Although the internal state that the kernel maintains on behalf of processes can be different across different kernels, the high-level properties of the process are much less likely to change. We capture the state of a process in terms of higher-level semantic information specified in the intermediate format rather than kernel specific data in native format to keep the format portable across different kernels. For example, the state associated with a Unix socket connection consists of the directory entry of the Unix socket file, its superblock information, a hash key, and so on. It may be possible to save all of this state in this form and successfully restore on a different machine running the same kernel. But this representation of a Unix socket connection state is of limited portability across different kernels. A different high-level representation consisting of a four tuple, virtual source pid, source fd, virtual destination pid, destination fd is highly portable. This is because the semantics of a process identifier and a file descriptor is typically standard across different kernels, especially across minor version differences.

The intermediate representation format used by Pea-Pods for migration is chosen such that it offers the degree of portability needed for migrating between differ-

ent kernel minor versions. If the representation of state is too high-level, the checkpoint-restart mechanism could become complicated and impose additional overhead. For example, the Pea-Pod system saves the address space of a process in terms of discrete memory regions called VM areas. As an alternative, it may be possible to save the contents of a process’s address space and denote the characteristics of various portions of it in more abstract terms. However, this would call for an unnecessarily complicated interpretation scheme and make the implementation inefficient. The VM area abstraction is standard across major Linux kernel revisions. Pea-Pods view the VM area abstraction as offering sufficient portability in part because the organization of a process’s address space in this manner has been standard across all Linux kernels and has never been changed since its inception.

Pea-Pods further support migration across different kernels by leveraging higher-level native kernel services to transform intermediate representation of the checkpointed image into an internal representation suitable for the target kernel. Continuing with the previous example, Pea-Pods restore a Unix socket connection using high-level kernel functions as follows. First, two new processes are created with virtual PIDs as specified in the four tuple. Then, each one creates a Unix socket with the specified file descriptor and one socket is made to connect to the other. This procedure effectively recreates the original Unix socket connection without depending on many kernel internal details.

This use of high-level functions helps in general portability of using Pea-Pods for migration. Security patches and minor version kernel revisions commonly involve modifying the internal details of the kernel while high-level primitives remain unchanged. As such services are usually made available to kernel modules through exported kernel symbol interface, the Pea-Pod system is able to perform cross-kernel migration without requiring modifications to the kernel code.

The Pea-Pod checkpoint-restart mechanism is also structured in such a way to perform its operations when processes are in a state that checkpointing can avoid depending on many low-level kernel details. For example, semaphores typically have two kinds of state associated with each of them: the value of the semaphore and the wait queue of processes waiting to acquire the corresponding semaphore lock. In general, both of these pieces of information have to be saved and restored to accurately reconstruct the semaphore state. Semaphore values can be easily obtained and restored through GETALL and SETALL parameters of the semctl system call. But saving and restoring the wait queues involves manipulating kernel internals directly. The Pea-Pod mechanism avoids having to save the wait queue information by requiring that all the processes be stopped before taking the checkpoint. When a process waiting on a semaphore receives a stop signal, the kernel immediately releases the process from the wait queue and returns

EINTR. This ensures that the semaphore wait queues are always empty at the time of checkpoint so that they do not have to be saved.

While Pea-Pods can abstract and manipulate most process state in higher-level terms using higher-level kernel services, there are some parts that not amenable to a portable intermediate representation. For instance, specific TCP connection state like timestamp values and sequence numbers, which do not have a high-level semantic value, have to be saved and restored in order to maintain a TCP connection. As this internal representation can change, its state needs to be tracked across kernel versions and security patches. Fortunately, there is usually an easy way to interpret such changes across different kernels because networking standards such as TCP do not change often. Across all of the Linux 2.4 kernels, there was only one change in TCP state that required even a small modification in the Pea-Pod migration mechanism. Specifically, in the Linux 2.4.18 kernel, an extra field was added to TCP connection state to address a flaw in the existing syncookie mechanism. If configured into the kernel, syncookies protect an Internet server against a synflood attack. When migrating from an earlier kernel to Linux-2.4.18, the Pea-Pod system initializes the extra field in such a way that the integrity of the connection is maintained. In fact, this was the only instance across all of the Linux 2.4 kernel versions where an intermediate representation was not possible and the internal state had changed and had to be accounted for.

To provide proper support for Pea-Pod virtualization when migrating across different kernels, we must ensure that any changes in the system call interfaces are properly accounted for. As pea-pods have a virtualization layer using system call interposition mechanism for maintaining namespace consistency and ensuring pea security, a change in the semantics for any system call intercepted by pea-pods could be an issue in migrating across different kernel versions. But such changes usually do not occur as it would require that the libraries be rewritten. In other words, Pea-Pod virtualization is protected from such changes in a similar way as legacy applications are protected. However, new system calls could be added from time to time. Such system calls could have implications to the pea encapsulation mechanism. For instance, across all Linux 2.4 kernels, there were two new system calls, `gettid` and `tkill` for querying the thread identifier and for sending a signal to a particular thread in a thread group, respectively, which needed to be accounted for to properly virtualize Pea-Pods across kernel versions. As these system calls take identifier arguments, they were simply intercepted and virtualized.

## 5 Security Analysis and Examples

Saltzer and Schroeder[37] describe several principles for designing and building secure systems. These include:

- *Economy of mechanism:* Simpler and smaller systems

are easier to understand and ensure that they do not allow unwanted access.

- *Fail safe defaults:* Systems must choose when to allow access as opposed to choosing when to deny.
- *Complete mediation:* Systems should check every access to protected objects.
- *Least privilege:* A process should only have access to the privileges and resources it needs to do its job.
- *Psychological acceptability:* If users are not willing to accept the requirements that the security system imposes, such as very complex passwords that the users are forced to write down, security is impaired. Similarly, if using the system is too complicated, users will misconfigure it and end up leaving it wide open.
- *Work factor:* Security designs should force an attacker to have to do extra work to break the system. The classic quantifiable example is when one adds a single bit to an encryption key, one doubles the key space an attacker has to search.

Pea-Pods are designed to satisfy these six principles. Pea-Pods provide economy of mechanism using a thin virtualization layer based on system call interposition and file system stacking that only adds a modest amount of code to a running system. The largest part of the system is due to the use of a null stackable file system with 7000 lines of C code, but this file system was generated using a simple high-level file system language [45], and only 50 lines of code were added to this well tested file system to implement the Pea-Pod file system security. Furthermore, Pea-Pods change neither applications nor the underlying operating system kernel. The modest amount of code to implement Pea-Pods makes the system easier to understand. Since the Pea-Pod security model only provides resources that are explicitly stated, it is relatively easy to understand the security properties of resource access provided by the model.

Furthermore, Pea-Pods provide fail safe defaults by only providing access to resources that have been explicitly given to peas and pods. Since Pea-Pod virtualization limits access to the underlying system to its virtual namespace, Pea-Pods provide complete mediation to operating system resources. Peas in pods are explicitly designed to provide least privilege by restricting programs in an environment that can be easily limited to provide the least amount of access for the encapsulated program to do its job. Pea-Pods provide psychologically acceptability by providing users and system administrators with a standard system environment where all they have to understand are their applications and the system resources that they need without detailed understanding of any underlying operating system specifics.

Similar to least privilege, Pea-Pods increase the work factor that it would take to compromise a system by simply not making available the resources that attackers depend on to harm a system once they have broken in. For example, since Pea-Pods can provide selective access to what program are included within their view, it would be very difficult to get a root shell on a system that does not have access to any shell program. Similarly, the fact that one can migrate a system away from a host that is vulnerable to attack increases the work an attacker would have to do to make services unavailable.

We briefly describe three examples that help illustrate how Pea-Pods can be used to improve computer security and application availability for different application scenarios. The application scenarios are e-mail delivery, web content delivery, and desktop computing.

For e-mail delivery, Pea-Pods can isolate different components of e-mail delivery to provide a significantly higher level of security in light of the many attacks on Sendmail vulnerabilities that have occurred. Consider isolating a Sendmail installation that also provides mail delivery and filtering via Procmail. E-mail delivery services are often run on the same system as other Internet services to improve resource utilization and simplify system administration through server consolidation. However, this can provide additional resources to services that do not really need them, potentially increasing the damage that can be done to the system if attacked. Using Pea-Pods, both Sendmail and Procmail can execute in the same pod, which isolates e-mail delivery from other services on the system. Since pod's allow one to migrate a service between machines, the e-mail delivery pod is migratable. If a fault is discovered in the underlying host machine, the e-mail delivery service can be moved to another system while the original host is patched, preserving the availability of the e-mail service.

Furthermore, Sendmail and Procmail can be placed in separate peas which facilitate necessary interprocess communication mechanisms between them while improving isolation. This pod is a common example of a privileged service that has child helper applications. In this case, the Sendmail pea is configured with full network access to receive e-mail, but without shell access since there is no reason why Sendmail needs a shell. Sendmail would be denied write access to file system areas such as `/usr/bin` to prevent modification to those executables, and would only be allowed to transition a process to the Procmail pea if it is executing Procmail. On mail delivery, Sendmail would then exec Procmail in the Procmail pea, which would be configured with more liberal access to process shell scripts and run other programs such as SpamAssassin. As a result, the Sendmail/Procmail pod can provide full e-mail delivery service while isolating Sendmail such that even if Sendmail is compromised by an attack, such as a buffer overflow, the attacker would be contained in the Sendmail pea and not even be able to execute a root shell to attempt to further

compromise the system.

Note that there are multiple ways to configure Internet services peas. With the e-mail delivery example, we illustrated a simple system configuration to prevent the common buffer overflow exploit of getting the privileged server to execute a local shell. By simply denying access to shells but allowing access to other files, we limit the amateur attacker's ability to exploit flaws, while requiring very little configuration or knowledge of the actual services. On the other hand, one can also use Pea-Pods to create a complete least privilege environment to contain more professional attackers to the domain they exploited.

For web content delivery, Pea-Pods can isolate different components of web content delivery to provide a significantly higher level of security in light of common web server attacks that may exploit CGI script vulnerabilities. Consider isolating an Apache web server front end, a MySQL database backend, and CGI scripts that interface between them. While one could run Apache and MySQL in separate pods, since they are providing a single service, it makes sense to run them within a single pod that can be migrated as a unit. If the underlying host comes under attack, such as via a denial of service attack, one can use the pod's migration mechanism to move the web content delivery pod to a safer machine, providing better service availability in a hostile environment. However, since both Apache and MySQL are within the pod's single namespace, if an exploit is discovered in Apache, it could be used to perform unauthorized modifications to the MySQL database.

To provide greater isolation among different web content delivery components, we can use three peas in a pod: one for Apache, a second for MySQL, and a third for the CGI programs. Each pea is configured to contain the minimal set of resources needed by the processes running within the respective pea. The Apache pea includes the apache binary, configuration files and the static html content, as well as a rule to exec all CGI programs into the CGI pea. The CGI pea contains the relevant CGI programs as well as access to the MySQL daemon's named socket, allowing interprocess communication with the MySQL daemon to perform the relevant SQL queries. The MySQL pea contains the mysql daemon binary, configuration files and the files that make up the relevant databases. Since Apache is the only program exposed to the outside world, it is the only process that can be directly exploited. However, if an attacker is able to exploit it, the attacker is limited to a pea that is only able to read or write specific Apache files, as well as exec specific CGI programs into a separate pea. Since the only way to access the database is through the CGI programs, the only access to the database an attacker would have is what is allowed by said programs. Consequently, it becomes very difficult to cause serious harm to such a Pea-Pod web content delivery system.

For desktop computing, Pea-Pods enable desktop computing environments to accommodate mobile users across

separate administrative domains. As users move from one geographic location to another, Pea-Pods allow them to take their computing with them in a hassle-free way. Since Pea-Pods provide complete mediation as well as fail safe defaults, system administrators can allow desktop computing pods from separate security domains to migrate onto their hosts, since the processes within the pod are prevented from harming it and can be configured to only access files from the pod file system securely exported to remote machines via NFS over IPSec. Peas can also be used within the context of such a desktop computing environment to provide additional isolation. Many application used on a daily basis, such as mp3 players and web browsers, have had security holes in the past that could possibly enable attackers to cause them to execute malicious code or give them access to the entire local file system [19, 20].

To secure an mp3 player, an mp3 player pea can be created within a desktop computing pod that restricts the mp3 player's ability to make use of files outside of a special mp3 directory. Since most users store their music within its own subtree, this isn't a serious restriction. Most mp3 content should not trusted, especially if one is streaming mp3s from a remote site. By running the mp3 player within this fully restricted pea, a malicious mp3 cannot compromise the user's desktop session. This mp3 player pea is simply configured with three file system rules. A path specific rule that provides access to the mp3 player itself is required to load the application. A directory default rule that provides access to the entire mp3 directory subtree is required to give the process access to the mp3 file library. Finally, a path specific rule that provides access to the `/dev/dsp` audio device is required to allow the process to actually play audio.

To secure a web browser, a web browser pea can be created within a desktop computing pod that restricts the web browser's access to system resources. Consider the Mozilla web browser as an example. A Mozilla pea would need to have all the files Mozilla needs to run accessible from within the pea. Mozilla dynamically loads libraries itself and stores them along with its plugins within the `/usr/lib/mozilla` directory. By providing a directory default rule that provides access to that directory, as well as another directory default rule that provides access to the user's `.mozilla` directory, the Mozilla web browser can run as normal within this special Mozilla pea. One would also want the ability to be able to download and save files, as well as launch viewers, such as for postscript or mp3 files, directly from the web browser. This involves a simple reconfiguration of Mozilla to change its internal `application.tmp_dir` variable to be a directory that is within the Mozilla pea. By creating such a directory, such as `downloads` within the users home directory, and providing a directory default rule allowing access, we enable one to explicitly save files, as well as implicitly save when one wants to execute a helper application. Similarly,

Name	Description	Linux
getpid	average <code>getpid</code> runtime	350 ns
ioctl	average runtime for the FIONREAD ioctl	427ns
shmget-shmctl	IPC Shared memory segment holding an integer is created and removed	3361 ns
semget-semctl	IPC Semaphore variable is created and removed	1370 ns
fork-exit	process forks and waits for child which calls exit immediately	44.7 us
fork-sh	process forks and waits for child to run <code>/bin/sh</code> to run a program that prints "hello world" then exits	3.89 ms
Apache	Runs Apache under load and measures average request time	1.2 ms
Make	Linux Kernel compile with up to 10 process active at one time	224.5s
Postmark	Use Postmark Benchmark to simulate Sendmail performance	.002s
MySQL	"TPC-W like" interactions benchmark	8.33s

Table 2: Application Benchmarks

just like Mozilla is configured to run helper applications for certain file types, one would have to configure the Mozilla pea to execute those helper applications within their respective peas. As shown for an mp3 player, configuring such a pea for these process is fairly simple. The only addition one would have to make is to provide an additional pea transition rule to the Mozilla pea that tells the Pea-Pod system to transition the process to a separate pea on execution of programs such as the `mpg123` mp3 player or the `gv` postscript viewer.

## 6 Experimental Results

We implemented Pea-Pods as a loadable kernel module in Linux that requires no changes to the Linux kernel. We present some experimental results using our Linux prototype to quantify the overhead of using Pea-Pods on various applications. Experiments were conducted on a trio of IBM Netfinity 4500R machines, each with a 933Mhz Intel Pentium-III CPU, 512MB RAM, 9.1 GB SCSI HD and a 100 Mbps Ethernet connected to a 3Com Superstack II 3900 switch. One of the machines was used as an NFS server from which directories were mounted to construct the virtual file system for the Pea-Pods on the other client systems. The clients ran different Linux distributions and kernels, one machine running Debian Stable with a Linux 2.4.5 kernel and the other running Debian Unstable with a Linux 2.4.18 kernel.

To measure the cost of Pea-Pod virtualization, we used a range of micro benchmarks and real application workloads and measured their performance on our Linux Pea-Pod prototype and a vanilla Linux system. Table 2 shows the seven micro-benchmarks and four application bench-

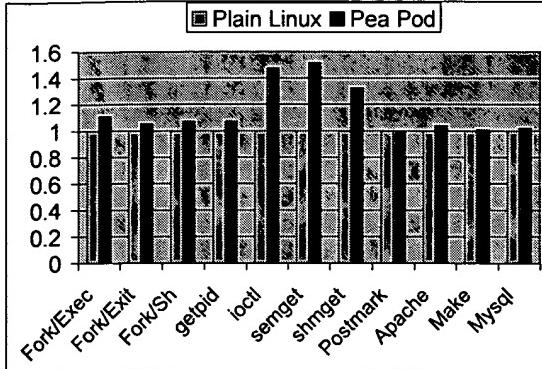


Figure 2: Pea-Pod Virtualization Overhead

marks we used to quantify Pea-Pod virtualization overhead as well as the results for a vanilla Linux system. To obtain accurate measurements, we rebooted the system between measurements. Additionally, the system call micro-benchmarks directly used the TSC register available Pentium CPUs to record timestamps at the significant measurement events. Each timestamp's average cost was 58 ns. The files for the benchmarks were stored on the NFS Server. All of these benchmarks were performed in a chrooted environment on the NFS client machine running Debian Unstable with a Linux 2.4.18 kernel. Figure 2 shows the results of running the benchmarks under both configurations, with the vanilla Linux configuration normalized to one. Since all benchmarks measure the time to run the benchmark, a small number is better for all benchmarks results.

The results in Figure 2 show that Pea-Pod virtualization overhead is small. Pea-Pods incur less than 10% overhead for most of the micro-benchmarks and less than 4% overhead for the application workloads. The overhead for the simple system call `getpid` benchmark is only 7% compared to vanilla Linux, reflecting the fact that Pea-Pod virtualization for these kinds of system calls only requires an extra procedure call and a hash table lookup. The most expensive benchmarks for Pea-Pods is `semget+semctl` which took 51% longer than vanilla Linux. The cost reflects the fact that our untuned Pea-Pod prototype needs to allocate memory and do a number of namespace translations. The `ioctl` benchmark also has high overhead, because of the 12 separate assignments it does to protect the call against malicious root processes. This is large compared to the simple FIONREAD `ioctl` that just performs a simple dereference. However, since the `ioctl` is simple, we see that it only adds 200 ns of overhead over any `ioctl`. For real applications, the most overhead was only four percent which was for the Apache workload, where we used the `http_load` benchmark [30] to place a parallel fetch load on the server with 30 clients fetching at the same time. Similarly, we tested MySQL as part of a web-commerce scenario outlined by TPC-W with a bookstore servlet running on top of Tomcat with a MySQL back-end. The Pea-Pod overhead for this scenario was less than 2%

Name	Applications
E-mail	Sendmail 8.12.3 with the pod configured to automatically change peas on execution of Procmail.
Web	Apache 1.3.26 and MySQL 3.23.49 running within separate peas inside the same Pod.
KDE	Xvnc – VNC 3.3.3r2 X Server KDE – Entire KDE 2.2.2 environment, including window manager, panel and assorted background daemon and utilities SSH – openssh 3.4p1 client inside a KDE konssole terminal connected to a remote host Shell – The Bash 2.05a shell running in a konsolle terminal KGhostView – A PDF viewer with a 450k 16 page PDF file loaded. Konqueror – A modern standards compliant web browser that is part of KDE KOffice – The KDE word processor and spreadsheet programs

Table 3: Application Scenarios for Migration

Case	Checkpoint	Restart	Size	Compressed
E-mail	0.079s	0.049s	848KB	124KB
Web	0.308s	0.508s	5.3MB	332KB
KDE	0.851s	0.942s	35MB	8.8MB

Table 4: Pea-Pod Migration Costs

versus vanilla Linux.

To measure the cost of Pea-Pod migration and demonstrate the ability of Pea-Pods to migrate real applications, we migrated the three application scenarios discussed in Section 5, an email delivery service using Sendmail/Procmail, a web content delivery service using Apache/MySQL, and a KDE desktop computing environment with an isolated web browser. Table 3 described the configurations of the application scenarios we migrated. To demonstrate our Pea-Pod prototype's ability to migrate across Linux kernels with different minor versions, we checkpointed each application workload on the 2.4.5 kernel client machine and restart it on the 2.4.18 kernel machine. For these experiments, the workloads were checkpointed to and restarted from local disk.

Table 4 shows the time it took to checkpoint and restart each application workload. In addition to these, migration time also has to take into account network transfer time. As this is dependent on the transport medium, we include the uncompressed and compressed checkpoint image sizes. In all cases, checkpoint and restart times were fast, taking less than a second for both operations, even when performed on separate machines or across a reboot. We also show that the actual checkpoint images that were saved were modest in size for complex workloads. For example, the KDE pod had over 30 different processes running, providing the desktop applications applications, as well as substantial underlying window system infrastructure, including inter-application sharing, a rich desktop interface man-

aged by a window manager with a number of applications running in a panel such as the clock. Even with all these applications running, they checkpoint to a very reasonable 35 MB uncompressed for a full desktop environment. Additionally, if one needed to transfer the checkpoint images over a slow link, Table 4 shows how they can be compressed very well with the bzip2 compression program.

## 7 Related Work

Historically, the military has been concerned with confidentiality and controlling the flow of information. Bell and LaPadula [8] as well as Biba [9] formulated models that formalize the concepts of ensuring confidentiality and integrity constraints between programs running at different classification levels. The work was incorporated into Multics' Multilevel Security Model [22] and the later Orange Book specification [14]. This work on information flow [24] is orthogonal to Pea-Pods, which focuses on containing untrusted applications.

Language-based tools have been used to try to harden the applications against buffer overflow attacks. Examples of this include the StackGuard compiler [13] and the Lib-Safe [6] interposition library. Similarly, others have strived to encourage the use of safer languages and language features, such as the type safety of ADA and Java. While LibSafe can work with unmodified dynamically linked applications, the majority of these solutions require applications to be rewritten or recompiled. Pea-Pods compliment these approaches by providing isolation of legacy applications without modification.

Privilege separation [32, 4] is a programming model that can be used to help prevent malicious code from executing in a privileged context. By separating each task of a system into a small process, one can create multiple simple programs that work together to perform a complex task and are easier to verify for correctness. Since the system is split into multiple processes, each process can be given a restricted set of privileges based on what it needs to do. OpenSSH and Qmail are two program examples that implement privilege separation. The Pea-Pod sandbox provides a form of privilege separation for legacy processes without requiring a redesign of the application service.

NSA's Security Enhanced Linux [26], which is based upon the Flask Architecture [40], implements a policy language that one can use to implement models that enable one to enforce privilege separation. The policy language is very flexible, but this also makes them very complex. Their example security policy is over 80 pages long. There is research into creating tools to make policy analysis tractable [2], but the fact that the language is so complex makes it difficult for the average end user to construct an appropriate policy. Peas, like NSA SE Linux, operate on a resource level where every resource is tagged, while Pod's operate like a virtual machine where resources not allocated to the

namespace are unavailable. Pods offer simplicity, such that even a novice administrator can determine what's available to both well behaved and malicious code. Peas provide the ability to provide simple increases in security, while also scaling up in complexity as required.

Janus [43, 17] and Systrace [31] are rule-based systems used for determining access controls. They implement system call interposition to control at an individual system call level what kernel functionality a process can use. Systrace provides graphical tools that help build rules on the fly. However, policy creation for Janus and Systrace requires a fine understanding of system calls. This provides great flexibility, but it makes them hard to configure, as well making final configurations difficult to understand. Like Pea-Pods, Janus and Systrace operate at the system call level. Unlike Pea-Pods, Janus and Systrace are also configured at the same individual system call level. Neither system integrates support for secure isolation with migration capabilities.

FreeBSD's Jail mode [21] implements a simpler to understand sandbox. It provides a chroot like environment that processes can not break out of. However, since Jail is limited in what it can do, such as the fact it doesn't allow IPC within a jail[16] many real world application will not work. Pea-Pods, on the other hand, do not place any restrictions on the types of applications that can run in its sandboxed environment.

SubDomain [12] creates a sandboxed view of the underlying file system for applications to run in. Like the pea-aware file system, it attempts to allow a system administrator to limit a processes' file system view to the minimum set needed by that application. However, since SubDomain's sandbox doesn't encapsulate processes, processes running as root can take advantage of system calls such as signal to affect change on processes outside their sandbox. While the Pea-Pods file system model is similar to SubDomain, it is conceptually different. While SubDomain operates at the system call level, the pea file system is a full-fledged file system. For example, when a file is opened, SubDomain must resolve it if it is a symbolic link. Pea-Pods, on the other hand, just uses the permission associated with the file at the end of the link as a regular file system does. Similarly, since Pea-Pods includes a full fledged file system, it integrates fully with the regular kernel security infrastructure and provides much better performance.

Virtual machine monitors (VMMs) can also be used to provide a secure sandbox environment [42, 44, 7]. VMMs can also be used to migrate an entire operating system environment [38]. Pea-Pods can compliment the functionality of VMMs. Unlike Pea-Pods, VMMs decouple processes from the underlying machine hardware, but tie them to an instance of an operating system. As a result, VMMs cannot migrate processes apart from that operating system instance and cannot continue running those processes if the operating system instance ever goes down, such as during

security upgrades. In contrast, Pea-Pods decouple process execution from the underlying operating system which allows it to migrate processes to another system when an operating system instance is upgraded. Similarly, VMMs just provide a single operating system namespace and lack the ability to isolate components within an operating system. If a single process in a VMM is exploitable, malicious code can make use of it to access and make use of the entire set of operating system resources. Since Pea-Pod's decouple processes from the underlying operating system and it's resulting namespace, they are natively able to limit the separate processes of a larger system to the appropriate resources needed by them.

Many systems have been proposed to support process migration, but not in the context of supporting application availability in the presence of operating system patches and upgrades. Several such research operating systems [34, 27, 3, 36, 15, 5, 11] rely on a single system image across all machines for process migration, in addition to the ability to forward many operations to the home node. They do not provide migration across independent commodity operating systems. Several user-space migration systems have been designed to run on commodity operating systems [25, 33, 29, 10]. These systems are primarily designed for long running scientific computations and cannot support processes that use many standard operating system services, such as IPC. TUI [39] provides support for process migration across machines running different operating systems and hardware architectures. Unlike Pea-Pods, TUI has to compile applications on each platform using a special compiler and does not work with unmodified legacy applications. Pea-Pods build on Zap [28], which supports transparent migration across systems running the same kernel version. Unlike Zap, Pea-Pods provide pod security and support for isolating processes inside of a pod. Furthermore, Pea-Pods support transparent migration across different minor kernel versions, which is essential for providing application availability in the presence of operating system security upgrades.

Pea-Pods can be used to improve the security of trusted computing systems[41, 18], which can enable the operating system and third parties to determine the identity of a program and if it's authorized to be executed. However, if a fault is discovered within a running trusted program, an attacker can make use of that fault to inject untrusted code into the system enabling access to the full set of resources. For example, Microsoft's X-Box, which runs a trusted operating system on trusted hardware, enforces a policy of only loading authorized games. However, buffer overflows in the code of trusted games have enabled users to load an untrusted Linux kernel and use the X-Box as a normal computer [1]. Pea-Pods can be used to limit the resources available to faulty trusted programs and thereby further limit an attacker's ability to compromise a trusted computing system.

## 8 Conclusions

The Pea-Pod system provides an operating system virtualization layer that decouples process execution from the underlying operating system. The virtualization layer supports two key abstractions for encapsulating processes, peas and pods. Pods provide lightweight sandboxes that mirror the underlying operating system environment, and peas provide fine-grain least privilege environments within pods. Together, peas and pods can isolate untrusted applications within sandboxes, preventing them from being used to attack the underlying host system or other applications even if they are compromised. The Pea-Pod sandboxes can be transparently migrated across machines running different operating system kernel versions. This enables security patches to be applied to operating systems in a timely manner with minimal impact on the availability of sandboxed application services. Pea-Pod secure isolation and migration functionality is achieved without any changes to applications or operating system kernels. We have implemented Pea-Pods in a Linux prototype and demonstrated how peas and pods can be used to improve computer security and application availability for a range of applications, including e-mail delivery, web servers and databases, and desktop computing. Our results show that Pea-Pods can provide easily configurable, secure migratable sandboxes that can run a wide range of desktop and server Linux applications in least privilege environments with low overhead.

## References

- [1] Anonymous. Technical Analysis of 007: Agent Under Fire save game hack. <http://www.xbox-linux.org/docs/007analysis.html>, Jul 2003.
- [2] M. Archer, E. Leonard, and M. Pradella. Towards a Methodology and Tool for the Analysis of Security-Enhanced Linux. Technical Report NRL/MR/5540—02-8629, NRL, 2002.
- [3] Y. Artsy, Y. Chang, and R. Finkel. Interprocess communication in charlotte. *IEEE Software*, pages 22–28, Jan 1987.
- [4] N. Associates. Privman - a library to make privilege separation easy. <http://opensource.nailabs.com/privman/>.
- [5] A. Barak and R. Wheeler. MOSIX: An Integrated Multiprocessor UNIX. In *Proceedings of the USENIX Winter 1989 Technical Conference*, pages 101–112, San Diego, CA, Feb. 1989.
- [6] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.
- [7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.
- [8] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report ESD-TR-74-244, Mitre Corp, Bedford, MA, May 1973.

- [9] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, Mitre Corp, Bedford, MA, 1977.
- [10] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, 1995.
- [11] D. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar 1988.
- [12] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *14th USENIX Systems Administration Conference (LISA 2000)*, New Orleans, LA, Dec. 2000.
- [13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.
- [14] Department of Defense. Trusted Computer System Evaluation Criteria (Orange Book). Technical Report DoD 5200.28-STD, Department of Defense, Dec. 1985.
- [15] F. Dougulis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, Aug. 1991.
- [16] FreeBSD Project. Developer’s handbook. [http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/developers-handbook/seccure-chroot.html](http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/seccure-chroot.html).
- [17] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proc. Network and Distributed Systems Security Symposium*, Feb. 2003.
- [18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, October 2003.
- [19] GOBBLES Security. Local/remote mpg123 exploit. [http://www.opennet.ru/base/exploits/1042565884\\_668.txt.html](http://www.opennet.ru/base/exploits/1042565884_668.txt.html).
- [20] GreyMagic Security Research. Reading local files in netscape 6 and mozilla. <http://sec.greymagic.com/adv/gm001-ns/>.
- [21] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May 2000.
- [22] P. A. Karger and R. R. Schell. MULTICS Security Evaluation: Vulnerability Analysis. Technical Report ESD-TR-74-193, Mitre Corp, Bedford, MA, June 1977.
- [23] B. LaMacchia. Personal Communication, Jan 2004.
- [24] C. E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3):247–278, Sept. 1981.
- [25] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin Madison Computer Sciences, Apr. 1997.
- [26] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.
- [27] S. J. Mullender, G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren. Amoeba a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.
- [28] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.
- [29] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of Usenix Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, Jan 1995.
- [30] J. Poskanzer. [http://www.acme.com/software/http\\_load/](http://www.acme.com/software/http_load/).
- [31] N. Provos. Improving Host Security with System Call Policies. In *12th USENIX Security Symposium*, Washington, DC, Aug. 2003.
- [32] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington, DC, August 2003.
- [33] J. Pruyne and M. Livny. Managing checkpoints for parallel programs. In *2nd Workshop on Job Scheduling Strategies for Parallel Processing (In Conjunction with IPPS ’96)*, Honolulu, Hawaii, Apr. 1996.
- [34] R. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating System Principles*, pages 64–75, Dec 1984.
- [35] E. Rescorla. Security holes... Who cares? In *Proceedings of the 12th USENIX Security Conference*, Washington, D.C., Aug. 2003.
- [36] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.
- [37] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Fourth ACM Symposium on Operating System Principles*, Oct. 1973.
- [38] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.
- [39] P. Smith and N. C. Hutchinson. Heterogeneous process migration: The Tui system. *Software – Practice and Experience*, 28(6):611–639, 1998.
- [40] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. of the Eighth USENIX Security Symposium*, Aug. 1999.
- [41] Trusted Computing Platform Alliance. TCPA main specification v1.1b. <http://www.trustedcomputing.org>.
- [42] VMware, Inc. <http://www.vmware.com>.
- [43] D. Wagner. Janus: an approach for confinement of untrusted applications. Master’s thesis, University of California, Berkeley, 1999.
- [44] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.
- [45] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

# Virtualizing Operating Systems for Seamless Distributed Environments<sup>1</sup>

*Tom Boyd and Partha Dasgupta*

Department of Computer Science and Engineering  
Arizona State University  
Tempe AZ, U.S.A.

## Abstract

*Applications and operating systems can be augmented with extra functionality by injecting additional middleware into the boundary layer between them, without tampering with their binaries. Using this scheme, we separate the physical resource bindings of the application and replace it with virtual bindings. This is called virtualization. We are developing a virtualizing Operating System (vOS) residing on top of Windows NT, that injects all applications with the virtualizing software.*

*The vOS makes it possible to build communities of systems that cooperate to run applications and share resources non-intrusively while retaining complete application binary compatibility. In this paper, we describe the structure, architecture and operation of the virtualizing Operating System supporting our virtualization concepts and methodologies*

**Keywords:** Parallel/distributed computing systems, API Interception.

## 1. Introduction

The promise of global, seamless distributed systems, constructed out of many autonomous workstations has not materialized. This paper presents a design and preliminary implementation towards making such a system possible within a set of uniformly administered machines.

There are three major challenges hindering the development of distributed operating systems that bring seamless distribution to the desktop. The first challenge is the magnitude of change required for enhancing or adding to any of the system's capabilities.

The second challenge is the unavailability of applications for such distributed operating systems. If applications have to be modified and/or rewritten to take advan-

tage of the distributed substrate and the distributed operating system, then the approach is doomed to fail.

The third challenge is the legacy nature of current systems and applications. Any changes to the operating system functionality leads to adding newer application programming interfaces (APIs). Few, if any, applications are rewritten to use the newer APIs.

The resolution to these challenges is through the *unobtrusive injection* of new functionality into existing systems. This approach requires no changes to the operating system or the existing application base, and yet endows the system with additional functionality that can be made as transparent or opaque to the end user as is necessary.

Using this approach, regular shrink wrapped applications can be run on regular standard operating systems, yet the underlying system can be a set of autonomous machines, providing a seamless distributed environment.

### 1.1 Computing Communities

Our research is part of a larger project called “*Computing Communities*” (or CC) [1]. The goal of the CC project is to enable a group of computers to behave like a large community of systems. The community grows or shrinks based on dynamic resource requirements through the scheduling and moving of processes, applications and resource allocations between systems—all transparently.

The computers participating in the CC utilize a standard operating system and run stock applications. The key technique to achieve such a system is the creation of a “*virtualizing Operating System*” or vOS. The main theme in the vOS is of course “virtualization”, which is the decoupling of the application process from its physical environment. That is, a process runs on a virtual processor with connections to a virtual screen and virtual keyboard, using virtual files, virtual network connections, and other virtual resources. *The vOS has the ability to change the connections of the virtual resources to real resources at any point in time, without support from the application.* The vOS implements the functionality to

<sup>1</sup> This research is partially supported by grants from DARPA/Rome Labs (F30602-99-1-0517), Intel, and Microsoft, and is part of the “*Computing Communities*” project, a joint effort between Arizona State University and New York University.

The *vOS* implements the functionality to virtualize the resources by controlling the mapping between the physical resources (seen by the operating system) and virtual handles (seen by the application). The virtualization provided by the *vOS* can provide a plethora of advantages:

- The users can move their virtual “home machines” at will, even for applications that are currently executing.
- A critical service running on machine  $M_1$  can be moved to machine  $M_2$  if  $M_1$  has to be cast away.
- Schedulers can control the complete set of resources.
- Applications can use resources, transparently aggregated from several machines. For example, a memory-intensive application can use memory in remote machines.

Our current work is based on the Windows 2000 operating system (but is extensible to any stock operating system). Windows 2000, like other operating systems, is structured such that the applications and the operating system contain a clearly delineated point of indirection, which is easily exploitable to add or interpose a layer of middleware.

The structure of the remainder of this report is as follows: Section 2 provides a general description of the virtualizing Operating System and the virtualization components. Section 3 describes the architecture of the system. The implementation of the system is described in section 4 with current status detailed in section 5. Section 6 describes related work and section 7 summarizes this work.

## 2. Virtualizing Operating System

The central mechanism that provides the features and benefits of this approach is the *virtualizing Operating System* or *vOS*. The main *vOS* theme is “virtualization”, which is the decoupling of the application process from its physical environment. The core of the *vOS* operation is the *virtualizing System Manager* (*vSM*), which is a central management facility providing global coordination and control services (figure 1). To ensure that the *vOS* is scalable and to reduce issues of *vOS* fault tolerance, multiple *vSMs* may operate as peers and coordinate activities between their respective domains.

The *vSM* is located at one place, anywhere in the network and performs global functions. It works with the *virtualizing Executive* (*vEX*), a system command and control component residing on each participating system. The *vEX* is a Windows NT service acting as the *vSM*'s local agent and proxy.

The *vEX* uses and manages local workstation resources in combination with an API wrapper tool and the *virtualizing Interceptor* (*vIN*) to capture and administer the workstation processes that participate in the CC.

The *vBUS* performs the communication function between the system components. It is designed to provide

support for different intercommunication requirements including message priorities, multicasting/broadcasting and point-to-point operations.

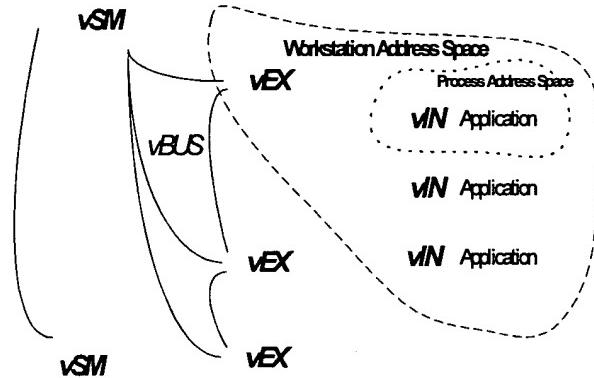


Figure 1: *vOS* system hierarchy

### 2.1 API/DLL Interception

As stated above, the *vIN* is responsible for capturing and then recording or reinterpreting much of the interactions between a running application and the underlying operating system. The power of virtualization comes from the reinterpretation of system calls and hence the capturing of system calls is a crucial underlying mechanism for our virtualization scheme. This capture is done by a scheme called *API Interception*.

API interception is gaining popularity in systems programming as it has unlimited potential for augmenting system functionality in a non-intrusive fashion. Most operating systems allow API interception methods to be built at the user level. In the Windows 2000 DLL scheme, when the application is loaded, the API references are resolved to a table of addresses in the user space called the *Import Address Table* (IAT), and filled in at run time [2]. The DLL contains a list of exported addresses used to populate the table. Using an indirect pointer, the application jumps to the API entry point within the DLL. By modifying the addresses contained in the IAT, the application call is redirected to an alternate API entry point.

### 2.2 Handle Virtualization

The Windows 2000 system is architected to use handles as references to most every component and resource of the system. The files, network and communications, processes, threads, fibers, events, windows, menus, sub-menus, edit buffers are just a few of the resources that have handles associated with them.

To virtualize applications and resources requires creating and mapping new handles and replacing references within API calls between systems (figure 2). Virtual handles allow each API to function correctly on the

local system as well as forming the basis for abstracting resource from specific system instances.

Handles normally consist of a 32-bit value. To aid in tracking and debugging, the handle is encoded with an origination code. The code includes an identifier for the source machine, process, thread and handle type. This information is useful for tracing or debugging a migrated process especially after several iterations.

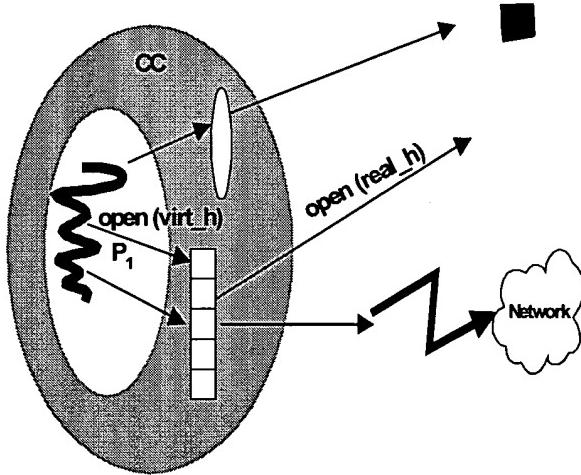


Figure 2: Handle virtualization

### 3. Architectural Overview

The main architectural components of the *vOS* are the *vSM*, *vEX*, *vIN*, and *vBUS*, as introduced in section 2. The following sections provide a review of the main architectural features of each of these components (figure 3).

#### 3.1 Virtualizing System Manager

The central component of the *vOS* is the *vSM*. It is a control process residing on any system within the *vOS* domain. The *vSM* is the primary interface with the user for system status reporting, command and control, system initialization and shutdown. The *vSM* is a window that reports current state and activities of the *vEXs*, *vINs* and system resources. The *vSM* carries the role of central and primary controlling agent and information source for the system. Each *vEX* communicates with the *vSM* to acquire knowledge of other *vEXs* and system resources. The *vSM* is located using the Windows 2000 Active Directory DNS service.

#### 3.2 Virtualizing Executive

Each system platform participating in the *vOS* contains a system executive (*vEX*) process. The *vEX* provides system level coordination and control. It acts as the common communication point for each of the *vIN* instances within the scope of one physical platform. A view of local and

remote resource, local system activities, security and policy are maintained and managed by the *vEX*.

*vEX* is a multithreaded NT service. Multiple threads handle the *vBUS*, command and control, local user interface, resource, migration, policy and failure management functions. The *vEX* is autonomous and quasi-persistent. If a local system failure occurs, the *vEX* can checkpoint its own, the *vINs* and the application's states and can migrate or be migrated elsewhere within the *vOS*. It performs this role by exchanging information periodically with the *vINs* and the *vSM*.

#### 3.3 Virtualizing Interceptor

The *vIN* is the interception middleware component. One *vIN* is required for each process or application participating in the *vOS*. *vIN* captures process information through the API interception mechanism. Several threads are established within the process state to handle: the *vBUS* IO, command and control, watchdog and trace and API interception. Each *vIN* communicates with the local system executive, *vEX*, for command and control directives. Virtualization tables<sup>2</sup> are built and maintained by the *vIN*. API calls creating, using or returning handles will always use a virtual handle created and maintained at this level. Where necessary it performs message marshalling, unmarshalling, forwarding and reception. Since process migration requires lower layer recreation or reassignment of handles, the virtual handle references are sent as part of the application's state information.

#### 3.4 Virtualizing BUS

The *vOS* is a single logical system constructed from multiple individual workstation components. Its overall performance depends upon effective interaction between the workstation instances. Efficient and timely sharing and exchange of information, status and messages flows between and throughout the *vOS* environment is important and is definable in terms of singular, group and universal relationships. The virtualizing BUS (*vBUS*) provides the support for the overall system operation within the context of the Windows NT and general Internet environment.

The virtualizing BUS is architecturally similar to a hardware bus in terms of the approach to the logical presentation and control of data. It uses a simple and efficient API for delivery and reception of *signals* and *messages*.

---

<sup>2</sup> *vOS* tables that contain the virtual to real handle relationships and other information such as handle or resource type and handle specific data such as addresses.

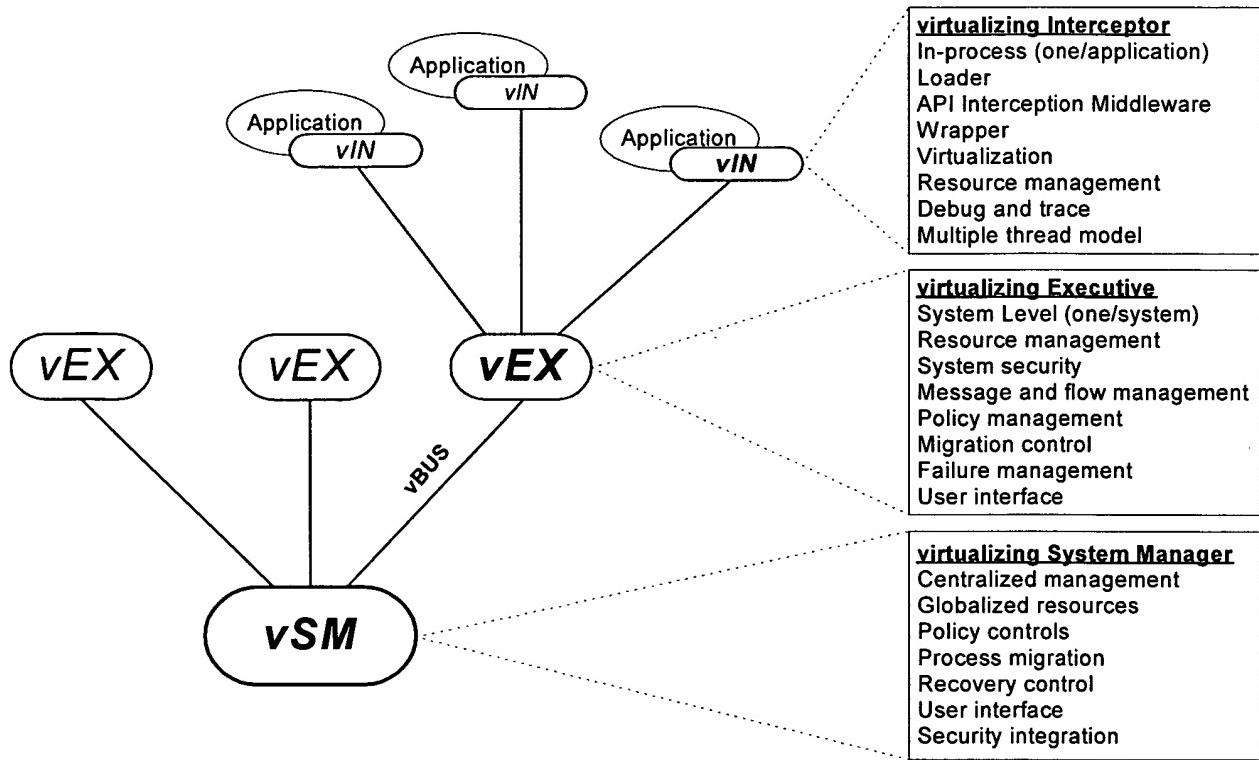


Figure 3: vOS Architecture

The architecture of the *vBUS* utilizes existing transport facilities: IP, TCP, UDP and other inherently available capabilities. With the introduction of RFC based multicast and broadcast protocol support, new opportunities are available to use facilities such Multicast Backbone technology [3].

#### 4. vOS System Archetype

We are building the virtualizing system using components and lessons learned from our prior work. The development proceeded from the initial development of the *vIN* as the foundation for the system. We next constructed the basic *vEX* using a simplified method to develop the *vEX/vIN* relationship.

##### 4.1 vSM Implementation

Upon initialization the *vSM*, presents the user interface window, initializes the *vBUS* and sets its service availability in the DNS. It waits until contact is made on a listening socket then accepts and begins communication with the contactor, which is a *vEX*. The *vEX* reports its status and resource information which is recorded in tables organized by *vEX* and *vIN* combinations. Resources are assigned currently using a persistent table, which can be altered through the user interface.

The user interface consists of a set of preset command selections and display areas. *vEX*, *vIN* and *vBUS*

information are available for display based on the selection made. Commands and control entries are selected from a static list. Additional information is requested depending upon the command. An automated periodic update request and display is available for a pseudo real-time update of the system status.

Currently, the *vSM* is manually launched. Since it will itself be migratable, we intend to build a more robust “homing service” that is capable of invoking and migrating the application without imbedding or tying the *vSM* to a single machine.

##### 4.2 vEX Implementation

The *vEX* is started whenever the base operating system becomes active. After activation, it initializes the *vBUS* and locates the *vSM* service through a DNS request. The *vSM* is contacted on a well-known port and the *vEX* sends basic state, location and resource information to the *vSM* then receives state, control and policy information.

The *vEX* next establishes the well-known file, event and mutex names for usage by the *vINs*. Once these names are established, the *vEX* waits for signals from the *vINs* and the *vSM*. Status is reported by the *vEX* upon request from the *vSM*. Commands from the *vSM* are executed and the results reported back to the *vSM*.

Impersonation<sup>3</sup> is used by the *vEX* and *vIN* when processes or resources are migrated. This allows the security for the currently logged user to be passed between the systems without requiring open use of logon or password information.

When a process or resource migration is requested, the receiving *vEX* starts a stub process which, the *vIN* recognizes and sets up to load and build the requested application environment. After establishing communication with the local *vEX*, the *vIN* receives the remote *vIN* address, contacts the remote *vIN* and proceeds to perform the requested migration task.

### 4.3 *vIN* Implementation

To capture the initialization state information for an application, *vIN* loads and executes at some point prior to the initial entry into the application. This is called process interdiction, and is implemented by intercepting the *CreateProcess()* and *CreateThread* API calls in *kernel32.dll*, setting the SUSPEND flag then injecting the *vIN* into the application space. After the interception environment is established, the process is resumed and the state information is collected.

Communication is established by the *vIN* to the *vEX* by opening a memory mapped file using a well known file name, signaling an event using a well known event name then waiting on a well known mutex access to the memory area. The *vEX* is waiting for this event signal. The common file area is used to communicate control values and some data between the processes. Once the well-known event is received by the *vEX*, a new unique memory mapped file name and event signal value is placed in the common area, and the well-known mutex is signaled. The *vIN* uses the file name and event signal to establish normal communications with the *vEX*.

As the application begins execution, the API initialization and continuous state values are captured, virtualized or stored. The virtualization values or references are placed into a structured table for later usage.

### 4.4 *vBUS* Implementation

To perform network I/O, the *vBUS* uses a library of Winsock 2 functions. Threads are used to support callbacks, signals, command and control, point-to-point transfers and broadcasts.

Callbacks are implemented as a function list, event code, class and type. When *vBUS* determines an event has occurred, it matches the event code and class with the function and performs the callback type. Signals are implemented using an assigned port number and a *select* function call. A *send/receive* pair passing *void* is used to create the signal event. Callback with *void* is used on the receive side to complete the signal.

<sup>3</sup> Impersonation is a mechanism in Windows 2000 to allow a process to run under another user's id (similar to "su" in Unix.)

Command and control, which is set as the highest priority activity thread in the *vBUS*, uses the callback facility to pass command notification to the *vBUS* instantiator. It also recognizes a limited set of commands for internal control.

Point-to-point transfers are buffer and forward operations. Data sent by the instantiator is sent to a receiver and data received is provided to the instantiator. Broadcasts are currently implemented using the *FCAST* code from Microsoft Research. Currently a Shutdown command is implemented which successfully causes the *vEX*s to become inactive and the *vIN*s to terminate.

## 5. Status

The status of the implementation is a set of prototypes that show the feasibility of the approach. Each of the above-mentioned components (*vSM*, *vEX*, *vIN* and *vBUS*) exists as separate programs with limited facilities and they have been tested to work together in several situations, described below:

A system that tests the overall concept of the *vOS* is our "window cloning" testbed [4]. This testbed uses a stock application called *RegMPad* (available from the MSDN Library), which is a multiple document interface variation of *Notepad*. The system is capable of intercepting and migrating parts of *RegMPad* such that the window and mouse controls are moved to a target machine and the logic execution happens on both the target machine and the source machine (window message processing and menu processing on the target machine and the rest remain on source machine).

We have also extensively experimented with process migration of single threaded processes. In [5] we show how to migrate a process that has active network connections, using our approach. In [6] we show how to migrate processes, which are actively interacting with users on the screen (using *WinMine*). Similar tests have been done with processes using files.

We are currently working on incorporating all the pieces of software that has been built into a coherent system with a clear delineation between the various components and the ability to interoperate and merge the features. We have targeted a multithreaded, network *telnet* application for migration. We are currently unraveling the multithreaded nature of Win32 storage assignment and usage.

## 6. Related Work

There are a few production and a number of research systems available today that use API/DLL redirection for interpositioning middleware. API interception can be done using toolkits such as *Detours* [7] or *Mediating Connectors* [8]. Systems using such facilities include:

**COP:** COP uses Detours and is a collaboration between Microsoft Research and the University of Rochester [9].

It is MFC oriented building and wrapping components around the Win32 API and using a COM interface for intersystem communication.

**NT-SwIFT:** *NT-SwIFT* [10] also known in its first release from Lucent Technologies as *SwIFT for Windows NT*, provides six functional components: automatic error detection and recovery, check pointing/message-logging, fault tolerance, event logging and replay, data replications and IP packet re-routing. It is capable of migrating applications between systems and restarting them in the event of a failure. It is assumed the server application can and should be modified to incorporate the SwIFT capabilities and that client applications do not necessarily need to be modified.

**Transparent Checkpoint Facility On NT:** This system performs API interception by rewriting the IAT to forward calls to the check pointing software, which is implemented as a DLL [11]. It captures system and application state while the application executes then replays the captured state data when restarting an application. Automatic check pointing is installed on the application by using an alternate loader and does not require application change, however, APIs are available for an application to explicitly control the system behavior. The system has several acknowledged limitations, including the requirement that temporary file state be retained for a restart, applications that bypass the IAT may not work correctly and multiple interacting processes are considered too complex to handle.

## 7. Summary

In this paper, we have presented an operating system in support of our virtualization work. It describes a system that uses the same fundamental unobtrusion philosophy as the virtualization techniques employed. The *vOS* is hierarchically structured to allow for system extensibility, but it still provides locally autonomy for robustness.

This system provides the basis for additional research into the viability and functionality of a Computing Community. Further work is in progress to incorporate security and session persistence as well as application adaptation, which includes fault detection and fault tolerance.

## 8. References

- [1] Partha Dasgupta, Vijay Karamchetti and Zvi Kedem, *Transparent Distribution Middleware for General Purpose Computations*, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 1999.
- [2] Jeffrey Richter, *Applications for Windows, Fourth Edition*, Microsoft Press, 1997.
- [3] Stephen Deering, *MBONE-The Multicast Backbone*, CERnet Seminar, 3 March 1993.
- [4] Tom Boyd and Partha Dasgupta, *Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization*, to appear in the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, June 2000.
- [5] Ravikanth Nasika and Partha Dasgupta, *Transparent Migration of Distributed Communicating Processes*, to appear in the 13<sup>th</sup> International Conference on Parallel and Distributed Computing Systems, August 2000.
- [6] Raghavendra Hebbalalu, *File Input/Output and Graphical Input/Output Handling for Nomadic Process on Windows NT*, Master's Thesis, Arizona State University, August 1999.
- [7] Galen Hunt and Doug Brubacher, *Detours: Binary Interception of Win32 Functions*, Proceedings of the 3<sup>rd</sup> USENIX Windows NT Symposium, July 1999.
- [8] Robert Balzer, *Mediating Connectors*, Proceedings of the 19<sup>th</sup> IEEE International Conference on Distributed Computing Systems Workshop, 1994, ISBN 0-262-57104-8.
- [9] Robert J. Stets, Galen C. Hunt, Michael L. Scott, *Component-based APIs: A Versioning and Distributed Resource Solution*, IEEE Computer, 32(7), July 1999.
- [10] Yennun Huang, P. Emerald Chung, Chandra Kintala, Chung-Yih Wang De-Ron Liang, and De-Ron Liang, *NT-SwIFT: Software Implemented Fault Tolerance for Windows NT*, 2nd USENIX Windows NT Symposium, July 1998.
- [11] Johny Srouji, Paul Schuster, Maury Bach, Yulik Kudmin, *A Transparent Checkpoint Facility on NT*, Proceedings of the 2<sup>nd</sup> USENIX Windows NT Symposium, August 1998.

**Sponsor Acknowledgment:** Effort sponsored by the Defense Advanced Research Projects Agency and AFRL/Rome, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0517. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

**Sponsor Disclaimer:** The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, AFRL/Rome, or the U.S. Government.

# Preemptive Module Replacement Using the virtualizing Operating System<sup>1</sup>

Tom Boyd

Department of Computer Science and Engineering  
Arizona State University  
Tempe AZ, U.S.A.  
tboyd@asu.edu

Partha Dasgupta

Department of Computer Science and Engineering  
Arizona State University  
Tempe AZ, U.S.A.  
partha@asu.edu

## ABSTRACT

Software aging causes software programs to fail over time. Rejuvenation of the software is a preemptive methodology developed to reduce failure, which reduces the need for complex methods to identify and fix problems after a failure has occurred. It does not eliminate the need for managing failure, it simply moves the bulk of the processing to a more controllable and simpler pre-failure state.

We have developed the virtualizing Operating System (vOS) residing on top of Windows 2000. The vOS is a middleware application developed to create an abstraction layer between an application and the underlying operating system. The abstraction layer allows for the virtualization of resources that a COTS application uses. The approach to virtualizing the application is by injecting functionality into running applications. Using this scheme, we separate the physical resource bindings of the application and replace it with virtual binding, referred to as virtualization. We use this technique to migrate an active program between systems without the program's awareness or involvement. We describe how to extend this capability to provide for both preemptive module level refreshment and program restart without modifying the application or the operating system.

## General-Terms

Management, Design, Reliability.

## Keywords

Self Healing, Software Rejuvenation, Parallel/Distributed Computing Systems, Virtualization.

## 1. INTRODUCTION

Self Healing refers to the detection and correction of a software fault or failure after the problem has occurred. Although it is sometimes easy to detect a program failure it is very difficult to correct the specific problem affecting the program. The complexity of the failure may cause more time to be spent in problem determination than to actually restart the application. However, restarting the application causes disruptions to other system activities and ultimately to the end user.

Preventive measures reduce the need for more complex detection and correction methods. By preventing a large number of failures, the actual detection and correction can be reduced to a rollback and restart level rather than a more complex examination and correction approach. However, most COTS applications do not benefit from advances in the area of rollback and restart due in part to the complexity and propriety nature of the applications. Traditionally, it has not been considered necessary to endow applications such as word processors, spreadsheets, browsers and such with self healing capabilities. However, having the ability to self heal provides immense advantages for all applications, especially in mobile and distributed environments.

In our research into virtualizing operating systems, we have found the need to be able to migrate any running application, without access to its source code. This paper describes how we have been able to build migration facilities, by decoupling the application from its environment (virtualization). We then propose to extend these existing mechanisms to use in module level rejuvenation. Our approach extends a new service to legacy applications allowing them to participate in newer "seamlessly distributed" computing environments without modification to their code.

The virtualization mechanism depends upon API interception methodology. The APIs of an application are typically serviced by library routines. Library routines are connected to the application using a layer of indirection. This layer of indirection presents an opportunity to capture and modulate the API call through the modification of the API call parameters. These modifications include the ability to capture, modify and restore state information inherently available within the API calls as well as implementing a system to capture and restore an application's execution state information.

This research discusses both an implementation and the core work that enables existing applications to assume new and novel characteristics and behaviors. Specifically, we use virtualization and process migration technologies of the virtualizing Operating System to provide module level rejuvenation. We also describe how the vOS system can be used to monitor and refresh itself, thus reducing aging failure within the vOS.

<sup>1</sup> This research is partially supported by grants from NSF, DARPA/Rome Labs and AFOSR.

The structure of the remainder of this paper is as follows: Section 2 provides the motivation for the creation of the virtualizing Operating System. Section 3 describes architecture, components and operation of the vOS. A discussion of software aging and rejuvenation is in section 4 with preemptive module refresh covered in section 5. Our module level rejuvenation is described in section 6 with Section 7 describing the vOS implementation of this process. Section 8 discusses related work and section 9 provides the paper summary.

## 2. BACKGROUND

Our research is part of a larger project called “*Computing Communities*” (or CC) [1]. The goal of the CC project is to enable a group of computers to behave like a large community of systems. The community grows or shrinks based on dynamic resource requirements through the scheduling and moving of processes, applications and resource allocations between systems—all transparently.

The computers participating in the CC utilize a standard operating system and run stock applications. The key technique to achieve such a system is the creation of a *virtualizing Operating System* or vOS [2,3]. The main theme in the vOS is “virtualization”, which is the decoupling of the application process from its physical environment. That is, a process runs on a virtual processor with connections to a virtual screen and virtual keyboard, using virtual files, virtual network connections, and other virtual resources. *The vOS has the ability to change the connections of the virtual resources to real resources at any point in time, without support from the application.* The vOS implements the functionality to virtualize the resources by controlling the mapping between the physical resources (seen by the operating system) and virtual han-

dles (seen by the application).

Our current work is based on the Windows 2000 operating system, but is extensible to any stock operating system. Windows 2000, like other operating systems, is structured such that the applications and the operating system contain a clearly delineated point of indirection, which is easily exploitable to add or interpose a layer of middleware.

## 3. VIRTUALIZING OPERATING SYSTEM

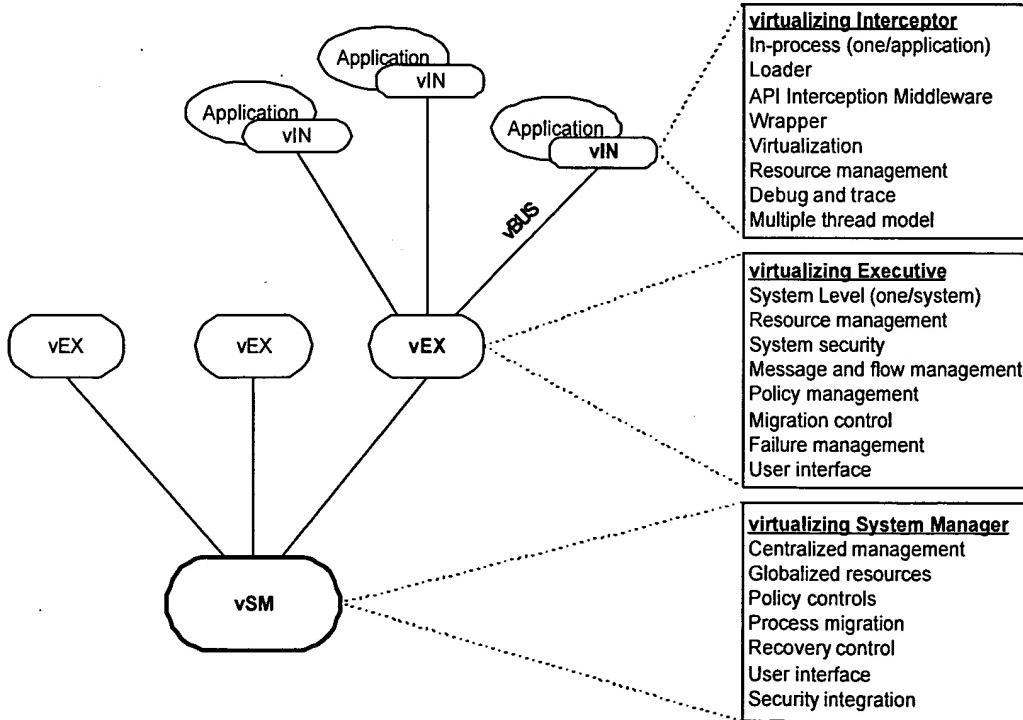
The *virtualizing Operating System* (vOS) is the central mechanism for our research. The main vOS theme is “virtualization,” which is the decoupling of the application from its physical environment.

### 3.1 Architecture

The vOS is a hierarchically structured and distributed application-management system that provides key global coordination and control services for the CC environment. Each vOS is responsible for a group of machines that becomes its bounded domain of control.

The vOS unobtrusively integrates its components into existing Windows 2000 system. It provides services that intercept and virtualize existing applications on the member machines without altering the application’s coding in any way.

The bounded domain of the vOS system is defined through the deployment of the three main system components (figure 1): the *virtualizing System Manager* (vSM), the *virtualizing Executive* (vEX) and the *virtualizing Interceptor* (vIN). Each of these components is placed at a key control point and is responsible for contributing to the overall functionality of the vOS. The control



points are at either the machine or application level.

### 3.2 Components

The *virtualizing System Manager* (*vSM*) acts as the central control program for the *vOS*. It is a process residing on a single workstation located anywhere within the network. The *vSM* manages individual workstations by interacting with a system service (a process) residing on each workstation named the *virtualizing EXecutive* (*vEX*). It displays the *vOS* system status and provides a command and control interface to each of the *vEX*s.

The *vEX* is located on each workstation. It uses the application virtualizer, named the *virtualizing INterceptor* (*vIN*), to capture and manage the individual workstation processes. The *vEX* consists of an application with a user interface that displays local system state and provides for local command and control activities.

The *vIN* is used by the *vEX* to initiate and capture process data from the workstation applications. The *vIN* is software injected into a running application, and it monitors the host application by intercepting its API calls to the underlying operating system. It resides in the same address space as the application under its control. It integrates a command interface into the application through the insertion of a menu item into the existing application menu, otherwise, it is completely unseen by the user of the interactive application.

The structure for this architecture is a tree with a single controlling element at the base (*vSM*), communicating with multiple individual systems through a module residing on each system (*vEX*) that in turn communicates with a module assigned to manage an individual application (*vIN*).

### 3.3 Key Technologies

Two technologies are used to enable the operation of the *vOS*. The first is the API/DLL Interception, which is based on Richter's [4] description. API's are intercepted by changing the addresses contained within the Import Address Table (IAT). The IAT contains a set of pointers filled in at runtime that resolve to the API fulfillment address in the appropriate DLL. Using this approach new code is inserted between the application and the DLL. Once the API is intercepted, we capture procedure call information, such as the handle, and manipulate or change it as required.

The second technology is the creation of an abstraction layer between a running application and the underlying system through the creation and management of virtual handles. Virtual handles are values that we create to replace the original handles returned in the API by the system. The application saves and uses the replacement handles as it would the real handles. The virtual handles are connected to real handles at the time the application makes requests to the system. This approach allows an application to move between computers transparently by remapping the virtual handles to real handles when required. The application does not change nor is it aware of the underlying modifications.

These two technologies work hand in hand to collect, save and manage the API call data, reserving it in migration tables within the *vIN*. The tables become a portion of the data copied and restored during process migration or module rejuvenation.

### 3.4 Migration Models

In our research we determined that the task of migrating a process can be divided into three somewhat overlapping approaches based on the type of state information required. Many processes require that only a small or minimal amount of state information needs to be collected and restored; this is referred to as the Minimal State Migration Model [5]. More complex processes involving complexities such as network, file and threading require that a more complete collection and restoration of process state be undertaken. Either a Full State Migration Model is required or for more specific module level controls, an Enhanced Minimal State Migration methodology is used. The enhancements include handling of files and managing active network connections.

Nasika and Heballal [6,7] provided the pioneering work and Zhang, Khambatti and Dasgupta [8] extended the findings for the Minimal State Model methodology. Given an application, a certain minimal set of state elements combined with a restart/suspend technique is all that is required to correctly migrate the specific set of processes. This minimal set of elements consists of the "data" area, heap allocation and handles. Tests have shown that these three components will correctly allow the recreation of process state. If file IO is present, then methods of capturing and restoring the file state for applications are defined. If network connections are present then the middleware layer handles the connection redirection.

## 4. DECAY AND REJUVENATION

As software executes, it shows signs of decay or aging [9,10,11]. Decay is caused by several factors. The first is undetected design flaws that create faults and failures over time. If the program must execute for an extended period, as versus single short period task, errors begin to accumulate and failure ultimately occurs. Design flaw errors are entirely a software issue. Both poor initial designs as well as periodic code modifications are the primary culprits for causing aging to occur.

The second type of error is produced by hardware faulting. Hardware can induce the same effect as software aging. Depending on the type and degree of the fault, any prescribed software solutions may provide the same benefits. However, hardware may only be masked for a short period of time. Ultimate failures are unavoidable.

In either the software or hardware case, errors accumulate and if given enough time, will cause the program to fail. The most prescribed method to resolve software decay is software rejuvenation [11,12]. Software rejuvenation is the refreshing of program execution environment through the stopping, reloading and restarting of the program or one or more of its elements [13]. The granularity of the refreshing operation can be as fine as memory defragmentation or as large as periodically restarting the entire computer. In any case, some set of proactive actions are prescribed based on an appropriate methodology for the application.

Rejuvenation has the merit of preventing errors from occurring by reducing the amount of time that a program or one of its components actually executes. Yurcik and Doss [13] comment that rejuvenation does not, remove bugs, the approach merely dodges them very effectively. So rather than running a system for a one year and increasing the chance of a failure, why not run the system for one day, 365 times [14]. In other words, by restarting the system more frequently, fewer aging related failures will occur.

Although restarting a system periodically has a failure reduction benefit, there is a cost associated with the restart. Specifically, the cost is full system unavailability for some period of time. A more effective approach is to use some form of targeted rejuvenation. Specifically, one of the finer grain methodologies should reduce the unavailability costs.

Using the body as a model, each of the cells on the body is replaced on a periodic basis. The replacement does not cause any type of interruption or trauma. In this way, the body continually refreshes itself and reduces the possibility for failure. Thinking of the cells as programs components, this same conceptual approach can be used to target specific parts of a running system, for refreshment. The finer the targeting, the less noticeable the impact to the system caused by the refresh.

Most COTS systems are designed to be a single composite set of routines that the program thread(s) use to perform its task. The routines are not considered in isolation while they are executing even though the APIs reside in various modules. This makes refreshing at a component level a significant problem since the specific modules being used may not be known by the application. Using the *vOS*, we are able to isolate to a module level, the APIs in use at any given time. Module level is sufficient for replacement to be the least disruptive.

## 5. PREEMPTIVE MODULE REFRESH

The methodology, frequency and timing of software refreshment can be based on a number of approaches. In general, however, some form of preemptive replacement approach is recommended. By replacing some targeted portion of the software environment on a schedule, static frequency or algorithmically approach, the desired reduction in failures is achieved. Studies are currently working on the optimal type of approach and methodology.

One of the strategies for performing component or system refreshing is at a macro system level focusing on hardware and the operating system. The IBM approach uses clustering in conjunction with failover to perform a system level movement of an application from one cluster node to another [12]. Then the operating system from the vacated node is refreshed by rebooting. This method resolves operating system aging using the existing cluster failover algorithms. At a finer level, this approach is also able to restart an individual service. However, the restarted service must be able to handle the restart by saving and restoring the appropriate data and state information.

Gupta and Jalote [15] suggest using a reactive technique of a combination of rollback with an online change when a fault occurs. The approach requires check pointing to provide the rollback. They further suggest that a more desirable proactive approach is to correct the fault online before performing the restart. Waiting until a failure occurs is more of a backstop in fault tolerance and should be available to some degree if more active measures fail.

Garg, Huang, Kintala and Trivedi [16] recommend combining check pointing with rejuvenation, thus “reducing the amount of rollback” after a failure. The issue becomes one of when to perform the rejuvenation. We are not as concerned in this paper with when to rejuvenate, but are instead interested in the method of rejuvenation.

The approach we are recommending is finer grain than the system level, does not require hardware assistance and is not limited to specialized services or modified applications. We describe next a module level restart that avoids the expense of a clustering environment, the loss of time associated with a full system reboot and enables any COTS application to participate in the refresh process. Choosing a module level restart is more efficient and less of a performance impact. In addition, our restart approach requires no modifications to existing applications and uses the existing *vOS* capabilities.

Module level restart works by reloading a copy of the text and then reorganizing and reapplying any state information. There can be a few up to many modules associated with an active program. However, it is unlikely that the program is using every API that the module contains. In fact, although an examination of the program may indicate that it imports some number  $N$  APIs, it in fact uses some smaller number  $P$  of these APIs ( $P < N$ ) during execution. This means that although the entire module is a target for refreshing, not every possible API carried within the module is affected. If we compare this approach to the refreshing of the entire program and all of its associated modules, the time trade off can be significant.

Our approach is to perform preemptive module refresh using the facilities of the *vOS*. The *vOS* provides an ongoing recording and storing of the state information at an application level. It saves the state information for the APIs that the application is using. This provides the best backstop for refreshing only the required APIs.

## 6. MODULE LEVEL REJUVENATION

The *vOS* is a middleware application that is inserted between an application and its support modules (section 3). Each of the APIs that a program uses within the support module is known by the *vOS*. The *vOS* intercepts each API call the program makes and the *vOS* code is executed before and after the API code is called.

With the API/Module information, the *vOS* performs on demand migration of active programs between systems using the *vIN*. The same data collected by the *vIN* and used for performing the migration constitutes a checkpoint and is applied to the refreshing process. One of the main distinctions between a full migration and the module level refresh is that only selected modules are refreshed. This allows for a finer refresh granularity which in turn requires less time and is less noticeable to the user.

To perform the module level rejuvenation, the *vOS* follows the following steps:

1. Suspends the program thread(s)
2. Unloads and flushes the target module
3. Reloads the target module
4. Reorganizes the module/API virtual table entries
5. Resumes the suspended thread(s)

One of the benefits of the *vOS* approach is that it does not require a specific checkpoint be performed since the data is collected as the API calls are made, thus saving time.

In the case of the application, a modified Enhanced Partial State process migration is performed (section 3.4). Instead of terminat-

ing and restoring the application and all of its module states, it only restores a selected main program component. The partial state approach allows the application's internal data to be rebuilt, essentially reorganized, before applying the final state. This allows the main program module(s) to be refreshed with minimal overhead.

## 7. VOS REFRESH

We have shown how the module level rejuvenation procedure works using the data collected and maintained by the *vIN*. The structure of the *vOS* supports further capabilities that enable additional self healing functionalities. Each of the main *vOS* components, *vSM*, *vEX* and *vIN*, are capable of differing degrees of independent operation. The *vSM*, for example, can be executed on any machine and the *vEX*s will locate and join it. The *vEX* does not require a *vSM* to operate. The *vIN* once injected by a *vEX*, operates without *vEX* involvement. Given this independence, each of the components can be extended to monitor, detect and refresh the other components.

The same aging issues apply to the *vOS* as apply to any program and it should also be required to be rejuvenated as part of the ongoing refreshing process. The individual *vOS* modules can refresh themselves as required and as a backstop, the different *vOS* components can monitor each other and resolve any issues that arise as part of the refresh process.

Since module level refresh of the *vOS* components is used, the time required to perform the refresh should be low and thus minimal impact. The *vOS* is aware of the refresh and can be readily modified to perform the refresh cooperatively. The applications are completely unaware of the refreshing process and are not affected by the activities.

Rejuvenating the operating system remains a separate issue. Regardless of the operating system in use, the primary prescribed method of refresh is through a reboot. However, even though this is required for off the shelf systems and is time consuming, it is not required as frequently as is examined in Castelli, Harper, Heidelberger, Hunter, Trivedi, Vaidyanathan, and Zegger [12]. Since the applications are refreshed, the operating system is not required to be rebooted as frequently to resolve application aging.

## 8. RELATED WORK

Much of the current literature covers work related to the modeling and analysis of software rejuvenation. Our work examines the application of rejuvenation. We found work from both the industry and research sectors that apply rejuvenation technology.

BASE [17] uses an N-Version Programming replication technique combined with a layer of abstraction to hide implementation details of off-the-shelf services. It proactively uses software rejuvenation, recovering the replicas on a periodic basis and then restarting the service by "rebooting" it. Current state information is supplied by the replicas in an abstracted form and a consensus state is applied to the new copy. The thesis is that "concrete state" aging corruption is hidden through the abstracted view of the state. It uses conformance wrappers to capture state information. The service remains available to the system from one of the replicas. We have chosen to avoid the developmental overhead associated with the NVersion Programming approach. Although multiple copies of the service shows high reliability against soft-

ware aging, the management and control overhead limits the applicability to COTS applications.

IBM Director [18] is a product announced by IBM (January 22, 2001) that incorporates IBM Software Rejuvenation [12]. The software proactively "identifies and predicts pending software problems" then schedules a rejuvenation for the identified software. It normally reboots the server on a planned basis or refreshes at a service level. It does not have the capability of refreshing an individual unmodified application, which is our main focus.

WinFT [19] is a set of library routines designed for Win95 and Windows NT. The routines provide support for detection and restart of a failed process, rebooting of a hung or broken operating system, checkpoint/recovery and software rejuvenation. Since the library functions must be called directly from within the application, it does not apply to existing COTS type applications.

## 9. SUMMARY

In this paper, we have made a proposal for performing module level replacement using the capabilities of the virtualizing Operating System. The *vOS* is the main platform implementation of the Computing Communities project and forms the basis for implementing the core technologies. It is distributed in nature and is integrated into the existing Windows 2000 environment without altering either the applications or the operating system. We believe that COTS software does not typically share the benefits of rejuvenation to reduce the effects of aging.

Software aging has been identified as a significant source of system failures. The prescribed solution to aging is software refreshing or rejuvenation. Many of the solutions available are on a system or specialized program level of granularity. We have chosen to focus on COTS applications such as spreadsheets and word processors which form the bulk of the user desktop applications. User applications stand the most to benefit from some form of aging prevention. For example, the word processor used to create this paper, crashed twice during its creation. From past experience, these crashes could have been avoided by "rebooting" the word processor on a periodic basis. Otherwise, some form of periodic module refreshing while I worked would have most likely achieved the same results.

Our focus is on preventative maintenance as a method to reduce the need for complex forms of self-healing. The module replacement approach reduces both the overhead and resultant time that a user would perceive. We have described how the capabilities of the *vOS* are tailor made for performing both module level and system level rejuvenation. No modifications are required to either the application or the operating system for our approach to work. This significantly extends the capabilities of the unchanged application and reduces the costs associated with adding the capability to existing systems.

## 10. REFERENCES

- [1] Dasgupta, P., Karamcheti, V. and Kedem, Z. Transparent Distribution Middleware for General Purpose Computations, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 1999.

- [2] Boyd, T. and Dasgupta, P. Virtualizing Operating Systems for Seamless Distributed Environments, in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, vol. 2, November 2000, pp. 735-740.
- [3] Boyd, T. and Dasgupta P. Injecting Distributed Capabilities into Legacy Applications Through Cloning and Virtualization, in the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, vol. 3, June 2000, pp. 1431-1437.
- [4] Richter, J. Applications for Windows, Fourth Edition, Microsoft Press, 1999.
- [5] Boyd, T. and Dasgupta, P. Process Migration: A Generalized Approach using a Virtualizing Operating System, To be published in the 22nd International Conference on Distributed Computing Systems, Vienna, Austria, July 2002.
- [6] Nasika, R. and Dasgupta, P. Transparent Migration of Distributed Communicating Processes, to appear in the 13<sup>th</sup> International Conference on Parallel and Distributed Computing Systems, August 2000.
- [7] Hebbalalu, R. File Input/Output and Graphical Input/Output Handling for Nomadic Process on Windows NT, Master's Thesis, Arizona State University, August 1999.
- [8] Zhang, S., Khambatti, M. and Dasgupta, P. Process Migration through Virtualization in a Computing Community, 13<sup>th</sup> IASTED International Conference on Parallel and Distributed Computing Systems, August 2001.
- [9] Parnas, D.L. Software Aging In the proceedings of the 16th International Conference on Software Engineering, May 1994.
- [10] Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A. Does Software Decay? Assessing the Evidence from Change Management Data, IEEE Transactions on Software Engineering vol. 27, no. 1, Jan. 2001.
- [11] Huang, Y., Kintala, C., Kolettis, N. and Fulton, N.D. Software Rejuvenation: Analysis, Module and Applications, Proceedings of the 25th Symposium on Fault Tolerant Computer Systems, Pasadena, CA, June 1995.
- [12] Castelli, V., Harper, R.E., Heidelberger, P., Hunter, S.W. K., Trivedi, S., Vaidyanathan, K. and Zeggert, W.P., Proactive Management of Software Aging IBM Journal of Research and Development, Vol. 45 No. 2, 2001.
- [13] Yurcik, W. and Doss, D. Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration, IEEE Software, July/August 2001.
- [14] Berstein, L. (2002, May), A Software Engineering Course for Trustworthy Software, Available: <http://www.dacs.dtic.mil/awareness/newsletters/stn3-4/trustworthy.html>.
- [15] Gupta, D. and Jalote, P. Increasing Software Reliability through Rollback and On-line Fault Repair, In the Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems, Taipei, Taiwan, December 1997.
- [16] Garg, S. , Huang, Y., Kintala, C.M.R. and Trivedi, K.S. Minimizing Completion Time of a Program by Checkpointing and Rejuvenation, ACM SIGMETRICS 96, Philadelphia, PA, May 1996.
- [17] Rodrigues, R. Castro, M. and Liskov, B. BASE: Using Abstraction to Improve Fault Tolerance, Proceedings of the 18th Symposium on Operating Systems Principles (SOSP '01), Banff, Canada, October 2001.
- [18] Press Release (2002, May), IBM Delivers Next Generation of Self-Healing Management Tools, available: [http://www-916.ibm.com/press/prnews.nsf/jan/B20162597ECE8738852\\_569DC0050C6AE](http://www-916.ibm.com/press/prnews.nsf/jan/B20162597ECE8738852_569DC0050C6AE)
- [19] Carreira, J. Costa, D. and Silva, J.G. WinFT: Using Off-the-Shelf Computers on Industrial Environments, Proceedings of the 6<sup>th</sup> International Conference on Emerging Technologies and Factory Automation, Los Angeles, September, 1997.

U.S. POSTAGE  
5 CENTS  
JUN 19 2008  
U.S. POSTAGE  
5 CENTS  
JUN 19 2008

Alexandria, VA 22313-1450  
If Undeliverable, return in Ten Days

---

OFFICIAL BUSINESS  
PENALTY FOR PRIVATE USE, \$300

**AN EQUAL OPPORTUNITY EMPLOYER**

**RECEIVED**

JUN 03 2008

USPTO MAIL CENTER